Savina - An Actor Benchmark Suite

Enabling Empirical Evaluation of Actor Libraries

Shams Imam Vivek Sarkar

shams@rice.edu vsarkar@rice.edu

Department of Computer Science, Rice University, Houston, USA.

Abstract

This paper introduces the Savina benchmark suite for actor-oriented programs. Our goal is to provide a standard benchmark suite that enables researchers and application developers to compare different actor implementations and identify those that deliver the best performance for a given use-case. The benchmarks in Savina are diverse, realistic, and represent compute (rather than I/O) intensive applications. They range from popular micro-benchmarks to classical concurrency problems to applications that demonstrate various styles of parallelism. Implementations of the benchmarks on various actor libraries are made publicly available through an open source release. This will allow other developers and researchers to compare the performance of their actor libraries on these common set of benchmarks.

Categories and Subject Descriptors D.1 [*Programming Techniques*]: Concurrent Programming; F.2 [*Analysis of Algorithms and Problem Complexity*]: General

General Terms Concurrent Programming, Measurement, Performance

Keywords Actor Model, Benchmark Suite, Java Actor Libraries, Performance Comparison

1. Motivation

Concurrent programs have become the norm with the proliferation of multicore processors. The Actor Model (AM) of concurrency [1] has recently gained popularity, in part due to the success achieved by its flagship language - Erlang. The AM is based on asynchronous message passing and offers a promising approach for developing reliable concurrent systems. With the success of Erlang in production settings, the AM has catapulted into the mainstream and there has been a proliferation of the development of Actor frameworks in popular sequential languages like C/C++ (CAF[9], Act++ [25]), Smalltalk (Actalk [7]), Python (Stackless Python [44], Stage [5]), Ruby (Stage [38]), .NET (Microsoft's Asynchronous Agents Library [30], Retlang [34]). Scala brings Erlang style actor based concurrency to the JVM [16]. Since then, many actor libraries and frameworks have been implemented to permit actor-style programming in Java: Jetlang [35], GPars [41], Lift [52], Scalaz [21], Akka [53], Habanero-Java [23], FunctionJava [15], etc. Developers can now design scalable concurrent applications on the JVM using actor libraries that automatically take advantage of multicore processors.

It is common for researchers and developers to use benchmark suites to help choose among different implementations. Further, benchmarks help motivate language implementers to improve their implementations and calibrate the competitive advantages of their approach. While micro-benchmarks are useful, micro-benchmarks rarely reflect the behaviour of larger real-world applications. A standard benchmark suite that goes beyond micro-benchmarks and allows end users to compare different implementations and use the one that delivers the best performance for a given use-case is highly desired. Unfortunately, such a suite does not exist as yet for actor programming models. As a result, researchers and developers resort to writing their own benchmarks, which can create issues with the repeatability and reliability of reported results.

This paper presents Savina, a benchmark suite for actor-oriented programs developed in the Habanero Extreme Scale Software Research Project at Rice University[1]. In this work, we are interested in developing a standardized benchmark suite that represents various use-cases in actor-oriented programs and allows users to do an apples-to-apples comparison between different actor libraries. Such a suite provides implementers of high-performance actor libraries an understanding of what the various use-cases are. It simplifies the identification of the issues that need to be corrected from the benchmark results and allows them to optimize for it. The benchmarks in Savina are diverse, realistic, and represent

AGERE! 2014, October 20 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2189-1/14/10...$15.00.
http://dx.doi.org/10.1145/2687357.2687368

[1] The red Savina pepper is a cultivar of the Habanero chili.

compute intensive applications. They range from popular micro-benchmarks to classical concurrency problems to applications that demonstrate various styles of parallelism. Implementations of the benchmarks on various actor libraries are made publicly available through an open source release. This will allow other developers and researchers to compare the performance of their actor libraries on these common set of benchmarks.

The paper is organized as follows: in Section 2 we give a brief description of the benchmarks in the `Savina` suite. Section 3 discusses related work. We describe the three categories of micro-benchmarks, concurrency benchmarks and parallelism benchmarks in Section 4, Section 5, and Section 6, respectively. We describe our methodology for initial experimental results in Section 7. We summarize our conclusions and future work in Section 8.

2. Benchmarks Introduction

A benchmark suite for the evaluation of actor runtimes should be representative of multiple use-cases and portable to many systems. The use of actors is very diverse and a good benchmark suite should cover various important domains. The goal of this work is to define a benchmark suite, `Savina`, that can be used to compare the performance of actor-oriented libraries and languages. `Savina` benchmarks are designed to be easily ported across different actor libraries (Section 7). The `Savina` benchmark suite focuses on computationally intensive applications, and includes both numeric and non-numeric problems. `Savina` aims to identify a representative set of actor applications which display commonly used parallel patterns. It covers applications that include common concurrency problems, graph and tree navigation, linear algebra, and stencil computations. The applications are compute intensive, and perform no I/O operations in their kernels.

`Savina` is designed to be extended with new benchmarks to allow the suite to evolve and address currently uncovered domains. In addition to comparing the performance of various runtimes, the benchmarks allow the comparison of code and other additional features supported by an implementation. The primary performance metric that is output by each benchmark code is elapsed time (in milliseconds) for running the kernel body. The source code of the suite is made available for the purpose of: *a*) verifying what is actually being tested, *b*) porting the benchmarks to other actor languages and runtimes, *c*) allowing comparison of solutions for syntax and elegance, and *d*) enabling analysis of benchmarks to further study performance, and the impact of different features in different actor libraries. In addition, the results from running the suite provides end users with additional information that allows them to choose actor libraries based on the benchmarks which closely fits their own applications.

A brief description of the `Savina` applications is shown in Table 1, which includes the name of the benchmark, the abbreviation used to refer to the benchmark, the parallel / concurrency pattern represented by the benchmark, and the source of the benchmark. The list includes many well-known micro-benchmarks that are already de facto standards in actor-oriented models, such as Ping Pong, Chameneos, and Thread Ring, as well as many other benchmarks. The benchmarks are divided into three categories: *a*) micro-benchmarks, *b*) concurrency benchmarks, and *c*) parallelism benchmarks.

Micro-benchmarks Micro-benchmarks are simple benchmarks that involve simple logic dedicated to test specific features of the actor runtimes. As seen in Table 1, there are 8 well-known programs used to analyze actor languages or libraries in `Savina`. These are designed to measure overheads in message delivery, messaging throughput, concurrent mailbox implementation, actor creation and destruction.

Concurrency benchmarks The AM being a model of concurrent computation is a natural fit for exploiting concurrency in computations. The second set of benchmarks in `Savina` have 8 programs and include Bounded-Buffer problem, Readers and Writers problem, and Dining-Philosophers problem among others. This set is a first step away from micro-benchmarks and towards more realistic applications. It focuses on classical concurrency problems which involves correctly coordinating non-deterministic interactions among multiple actors.

Parallelism benchmarks Taking full advantage offered by a multicore machine requires the writing of parallel code. The final set of benchmarks include 14 programs and concentrates on parallelism. Parallelism in the applications is obtained by task decomposition to effectively utilize multicores. The decomposition needs to be converted into an actor-style computation. The benchmarks include a wide variety of computations that display pipeline parallelism, phased computations, divide-and-conquer style parallelism, master-worker parallelism, and graph and tree navigation. The programs in this set are larger than the programs from the previous two sets and represents more realistic parallel computations.

We are unaware of any other comprehensive benchmark suite for actor frameworks like `Savina`, and have designed it to be an extensible framework so that more benchmarks can be easily added in the future. The goal is to cover a wide variety of patterns in the benchmarks which will not only allow comparison of performance, but also programmability of the solutions based on features available in the actor frameworks being evaluated. We encourage researchers to contribute optimized versions of `Savina` benchmarks for existing as well as new actor frameworks into a community repository. We envision that the optimized versions of each `Savina` benchmark program will evolve over time with increased community contribution.

#	Name	Symbol	Feature or Pattern being measured	Source
1	Ping Pong	PINGPONG	Message passing overhead	Scala [42]
2	Thread Ring	THREADRING	Message sending; Context switching between actors	Theron [28]
3	Counting Actor	COUNT	Message delivery overhead	Theron [26]
4	Fork Join (throughput)	FJTHRPUT	Messaging throughput	JGF [12], Authors
5	Fork Join (actor creation)	FJCREATE	Ahead of time Actor creation and destruction	JGF [12], Authors
6	Fibonacci	FIB	Incremental Actor creation and destruction	Cardoso et al. [8]
7	Chameneos	CHAMENEOS	Contention on mailbox; Many-to-one message passing	Haller [17]
8	Big	BIG	Contention on mailbox; Many-to-Many message passing	BenchErl [4]
9	Concurrent Dictionary	CONCDICT	Reader-Writer concurrency; $O(1)$ operations	Authors
10	Concurrent Sorted Linked-List	CONCSLL	Reader-Writer concurrency; $O(N)$ operations	Shirako et al. [36]
11	Producer-Consumer with Bounded Buffer	BNDBUFFER	Multiple message patterns based on Join calculus	Hoare [18], Sulzmann [39]
12	Dining Philosophers	PHILOSOPHER	Inter-process communication; Resource allocation	Hoare [19]
13	Sleeping Barber	BARBER	Inter-process communication; State synchronization	Dijkstra [11]
14	Cigarette Smokers	CIGSMOK	Inter-process communication; Deadlock prevention	Patil [33], Wikipedia [47]
15	Logistic Map Series	LOGMAP	Synchronous Request-Response with non-interfering transactions	Authors ([29, 48])
16	Bank Transaction	BANKING	Synchronous Request-Response with interfering transactions	Authors
17	All-Pairs Shortest Path	APSP	Graph exploration; Phased computation	Authors
18	A-Star Search	ASTAR	Graph exploration; Message priority	Authors
19	NQueens first K solutions	NQUEENK	Divide-and-conquer style parallelism; Message priority	Authors
20	Recursive Matrix Multiplication	RECMATMUL	Divide-and-conquer style parallelism; Uniform load	Authors
21	Quicksort	QUICKSORT	Divide-and-conquer style parallelism; Non-uniform load	Authors
22	Radix Sort	RADIXSORT	Static Streaming Pipeline; Message batching	StreamIT [43]
23	Filter Bank	FILTERBANK	Static Streaming Pipeline; Split-Join Pattern	StreamIT [43]
24	Bitonic Sort	BITONICSORT	Static Pipeline; Round-robin message forwarding	StreamIT [43]
25	Sieve of Eratosthenes	SIEVE	Dynamic Streaming Pipeline; Non-uniform load	GPars [41]
26	Unbalanced Cobwebbed Tree	UCT	Tree exploration; Non-uniform load	Zhao and Jamali [54]
27	Online Facility Location	FACLOC	Dynamic Tree generation and navigation	Authors
28	Trapezoidal Approximation	TRAPEZOID	Master-Worker; Static load-balancing	Stage [5], SALSA[46]
29	Precise Pi Computation	PIPRECISION	Master-Worker; Dynamic load-balancing	Authors
30	Successive Over-Relaxation	SOR	4-point stencil computation	SOTER [45]

Table 1: List of `Savina` Benchmarks divided into three categories: 8 micro-benchmarks, 8 concurrency benchmarks and 14 parallelism benchmarks.

3. Related Work

Many actor benchmarks and benchmark suites have been designed and are currently being used for many different purposes, but none match our goals for a diverse set of compute intensive applications which display commonly used parallel patterns. `bencherl` is a publicly available scalability benchmark suite for applications written in Erlang [4]. In contrast to other benchmark suites, which are usually designed to report a particular performance point, `bencherl` aims to assess scalability, i.e., help developers to study a set of performance points that show how an application's performance changes when additional resources (e.g., CPU cores, schedulers, etc.) are added. The benchmark suite comes with an initial collection of parallel and distributed benchmarks. The Theron C++ concurrency library provides five actor microbenchmarks [27] with detailed performance analysis.

Cardoso et al. [8] also point out the need for a benchmarking suite and make significant advances in that direction. They focus on microbenchmarks (Thread Ring, Chameneos, and Fibonacci) to compare the performance of languages supporting Actor and Agent concurrent programming models. They compare the performance of a variety of languages such as Jason, 2APL, GOAL, Erlang, Akka, and ActorFoundry. Our goals are similar in that we want to enable cross language performance comparison, but in `Savina` we also introduce a wider variety of benchmarks. In this paper, we focus on performance comparison of JVM-based actor libraries.

PARSEC [6] is a benchmark suite created to drive the design of the new generation of multiprocessors and multicore systems. The benchmarks included in the suite represent emerging workloads that implement state-of-the-art algorithms. PARSEC is similar to Savina in the sense that it is largely automated, allowing users to create scripts that will run the benchmarks with the requested combinations of input parameters. The goal of the PBBS benchmarks is not only to compare runtimes, but also to be able to compare code and other aspects of an implementation [37]. Like Savina, the benchmarks in PBBS are designed to make it easy for others to try their own implementations, or to add new benchmark problems.

The nofib suite [32] started in the early 1990s as a collection of Haskell programs for benchmarking the implementation of the Glasgow Haskell Compiler. It has since evolved as a benchmark suite geared towards functional languages, oriented mostly towards improving implementations and providing performance comparisons. Due to the variety of benchmarks included, another goal of nofib has been to allow users of the language and a specific implementation to predict the performance of their own programs. Our goals are similar in that Savina can be used to compare various implementations of actors. Finally, there have also been attempts to compare programming languages by defining a set of benchmarks. The Computer Language Benchmarks Game [14] captures a broad set of languages, it compares over 20 programming languages on a set of 13 micro-benchmarks.

4. Micro-benchmarks

In this section we briefly describe the micro-benchmarks included in Savina. These are designed to measure overheads in message delivery, messaging throughput, concurrent mailbox implementation, actor creation and destruction.

4.1 Ping Pong

The first micro-benchmark is the Ping Pong benchmark in which two actors send each other messages back and forth. The benchmark tests the message passing overheads in the implementation for actors by measuring the average time between sending a message to an actor and receiving a response. The original version of the code was obtained from [42] and ported to use each of the different actor frameworks. It is configured to run using two concurrent actors, a single integer value is sent back and forth between the Ping and Pong actors. The value is decremented each time it is sent, until it reaches zero, whereupon the benchmark terminates. An important point about the Ping Pong benchmark is that at any point, only one actor is ever sending or receiving the message. Thus though the benchmark may seem to be concurrent, it is actually sequential. One way to optimize the performance of the actually this benchmark is to maintain two dedicated processing units for each of the actors even when the actor's mailbox queue is empty. This benchmark can be configured to run with different values of N, the number of pings between the two actors.

4.2 Thread Ring

The ThreadRing micro-benchmark comes from Theron [28]. An integer token message is passed around a ring of N connected actors. Each passing of the token from one actor to the next is called a hop and constitutes the sending of a message after decrementing the value of the token. Thus until the integer value of the token reaches zero, there is a repeated switch from actor to actor by the passing of the token. Like the Ping Pong benchmark, the ThreadRing benchmark is actually sequential as at any point, only one actor is ever sending or receiving the message. Unlike the Ping Pong benchmark, the strategy of maintaining dedicated processing units for each actor is not beneficial as there can be many participating actors. The key to gaining performance in the ThreadRing benchmark is minimizing the overheads of message sending and of context switching from one actor to another. This means that the benchmark is a good test of the raw efficiency of message sending as well as the overheads of context switching between many actor instances, since it removes all contention for resources except between the sending and receiving actors. This benchmark can be configured to run with different values of N.

4.3 Counting Actor

The Counting Actor micro-benchmark has also been ported from Theron [26]. The benchmark measures the time taken to send N successive messages to a Counter actor. The Counter actor increments a local integer counter every time it receives an increment message. Finally the accumulated value of the Counter is queried and returned. The messages are all sent in series by the same sender, a Producer actor. To minimize the overhead of millions of small memory allocations for each of the messages sent to the Counter actor, a single instance of increment message is used at the Producer actor. Since many messages are sent to the Counter actor, it also stress tests the mailbox implementation which needs to store the pending messages yet to be processed. Thus, the benchmark effectively measures the message delivery overhead in the actor implementation. This benchmark can be configured to run with different values of N.

4.4 Fork Join (throughput)

The Java Grande Forum Fork-Join benchmark [12] throughput variant measures messaging throughput, the rate at which messages are processed by the implementation. The application first creates K worker actors, and then sends each actor a total N messages in a round robin manner. Since there is only one sender of messages to the workers, the concurrent nature of the mailbox is not tested in this benchmark. Each actor does a minimal amount of work processing each message, each worker actor terminates after processing N messages. Like the Counting Actor benchmark, performance can be achieved in this benchmark by batch processing the messages sent to the worker actors. This benchmark can be configured to run with different values of K and N.

4.5 Fork Join (actor creation)

The Java Grande Forum Fork-Join benchmark [12] actor creation variant measures the time taken to create and destroy actor instances. The application creates millions of actors sequentially and sends each actor a single message. Each actor receives a single message, does a minimal amount of work processing that message (similar to the throughput variant), and then terminates. This measures the ability of the actor implementation to create and support millions of actors. It also measures the heap allocation ability of the system since millions of actors are allocated and deallocated. Since there are far fewer processing units on the system compared to actors created, the actor runtime also needs to deal with the overhead of tracking pending messages in each actor instance. These messages need to be scheduled for processing when a processing unit becomes available. This benchmark can be configured to run with different values of N, the total number of actors created at the start.

4.6 Fibonacci

The Fibonacci benchmark has been ported from Cardoso et al. [8]. It involves the recursive creation of a predefined number of actors using the naive recursive formula for computing fibonacci numbers. Each parent actor spawns two additional actors after receiving a wrapped integer message if the value is greater than 2. Otherwise, the *leaf* actor sends a result value of 1 back to the *non-leaf* sender. Each non-leaf actor waits for a message each from the two actors it spawned, sums the values in the messages, and forwards the results back to the parent. The computation ends when the original (root) actor has received its results and prints it. This benchmark also measures the heap allocation ability of the system since millions of short-lived actors are allocated and deallocated. Unlike the Fork-Join actor creation variant the actors are allocated and deallocated incrementally with most of the actors being short-lived whereas other actors (which are closer to the root) are longer lived. Extracting good performance from this benchmark can require an initial breadth-first strategy while scheduling actors, but then switching to a depth-first strategy to minimize the number of live actor instances in the computation. This benchmark can be configured to run with different values of N, the index of the fibonacci number to compute.

4.7 Chameneos

The Chameneos micro-benchmark measures the effects of contention on shared resources while processing messages. The original Scala implementation was obtained from the public Scala SVN repository [17]. The benchmark involves all *chameneos* constantly sending messages to a mall actor that coordinates which two *chameneos* get to meet. The mall actor decides which two chameneos get to meet by sending one of the chameneos the address of its new partner. The two chameneos exchange some internal state and return to the mall with the hope of meeting new partner chameneos. The mall actor terminates the computation when a pre-configured number of meetings have taken place between all the participating chameneos. Chameneos are usually active on, at least, all but one processing unit and are concurrently sending messages to the mall actor's mailbox. Adding messages into the mall actor's mailbox serves as a contention point and stress tests the concurrent mailbox implementation. The benchmark can be configured by C, the number of chameneos and N, the total number of meetings that take place at the mall.

4.8 Big

Big is a benchmark from the bencherl [4] suite that implements a many-to-many message passing scenario. Several actors are spawned, each of which sends a ping message to a random recipient actor. The recipient actor responds with a pong message to any ping message it receives. Since each actor is also potentially concurrently receiving messages from all the other actors, this benchmark also measures the effects of contention on the recipient actor's mailbox. The benchmark is parametrized by N, the number of actors and P, the number of pings sent by each actor. After a P pings have been sent by each actor, it notifies a sink actor. Meanwhile the actor stops sending new pings and waits on responding to any pings it receives with pongs. When all actors are done sending pings, the sink actor has notifications from all N actors and terminates the computation by terminating all the actors.

5. Concurrency Benchmarks

In this section we briefly describe the concurrency related benchmarks included in Savina. Actors being a model of concurrent computation are a natural fit for most of these benchmarks. The set focuses on classical concurrency problems which involves correctly coordinating non-deterministic interactions among multiple actors.

5.1 Concurrent Dictionary and Sorted Linked-List

In the ReadersWriters problem [10, 50], there are multiple entities accessing a resource, some reading and some writing. In addition, there is the natural constraint that no entity may access the resource for reading or writing while another process is in the act of writing to it. Pure actor solutions exhibit no concurrency, so this benchmark (and the next benchmark) assists actor-based solutions that enable reader-writer concurrency. We use the Concurrent Dictionary (CD) data structure benchmark to measure the performance of reader-writer concurrency support. We mix read (lookup by item) and write (key-value pair `put`) requests into the actor representing the CD data structure. The read and write operations in the CD benchmarks take $O(1)$ time, thus also measures any overhead encountered while introducing intra-actor parallelism. The benchmark is configured by N, the total number of requests made to the CD actor by W worker actors. An additional parameter, P is used to configure the percentage of requests that are write requests.

The `Concurrent Sorted Linked-List` (CSLL) benchmark measures the performance of adding elements, removing elements, and performing collective operations on a sorted linked-list. The implementation maintains a single actor responsible for handling requests to the linked-list data structure. All the operations are implemented using a sequential walk of the data structure from the head of the list. Thus, the read and write operations take $O(N)$ time, allowing greater benefits to be achieved by introducing intra-actor parallelism and overcoming tasking overheads. There are multiple worker actors requesting various operations on the linked-list and read requests are processed in parallel. The benchmark is configured by N, the total number of requests made to the CSLL actor by W worker actors. An additional parameter, P is used to configure the percentage of requests that are write requests.

5.2 Producer-Consumer with Bounded Buffer

The `Producer-Consumer with a Bounded-Buffer` is a classic example of a multi-process synchronization problem [18, 49]. In this problem, producers push work into the buffer as it is produced while consumers pull work from the buffer when they are ready. In an actor-based solution to this problem, producers, consumers and the buffer are modeled as actors. The buffer actor acts like a manager and needs to keep track of at least the following scenarios: *a*) whether the data buffer is full or empty; *b*) when consumers request work from an empty buffer, the consumers are put in a queue until work is available; *c*) when producers are ready to produce data and the buffer is full, the producers are put in a queue until space is available in the data buffer; *d*) notify producers to produce more work when space becomes available in the data buffer. The buffer actor needs to maintain additional queues for the available consumers and producers since there is no way to disable the processing of particular messages and this convolutes the logic of the buffer actor. The buffer actor is like the mall actor in the Chameneos benchmark (Section 4.7) since it is concurrently receiving messages from both producers and consumers, so results of this benchmark should also account for mailbox contention overhead. Actor-based solutions also have overhead from maintaining additional data structures (adding and removing elements in collections) for the available producers and consumers in the buffer actor. The benchmark is configured by the buffer size B, the number of producers P, and the number of consumers C.

5.3 Dining Philosophers

The `Dining Philosophers` problem is another classic synchronization problem [19]. There are N philosophers sitting around a circular table eating or thinking. The problem is that each philosopher needs two forks to eat, and there are only N forks, one between each two philosophers. The problems deals with resource allocation strategies to avoid deadlock, a state in which no progress is possible as none of the philosophers can eat. The implemented actor-based solution introduces an arbitrator actor to guarantee that a philosopher actor can only pick up both forks or none at all. This schemes avoids the possibility that one of the philosophers holds on to one fork for a long time while waiting for another fork to be released, thus creating a possibility of a deadlock. The arbitrator either accepts a philosopher request by allocating both forks. Otherwise, the philosopher's request to eat is rejected and the philosopher tries to eat again by sending another request to the arbitrator. Performance benefits can be gained by scheduling strategies that minimize the number of failed eat requests by philosophers. The solution is parametrized by N, the number of philosophers, and M, the number of rounds each philosopher gets to eat. The solution also prints out the total number of messages received by the arbitrator, smarter solutions can benefit by using strategies that reduce failed attempts to acquire forks.

5.4 Sleeping Barber

The `Sleeping Barber` problem [11, 51] is a classic inter-process communication and synchronization problem between multiple concurrent entities (actors). The problem is defined as a barber's shop where there is only one barber, one barber chair and a number of waiting chairs for the customers. When there are no customers the barber sits on the barber chair and sleeps. When a customer arrives he awakes the barber or waits in one of the vacant chairs if the barber is cutting someone else's hair. When all the chairs are full, the newly arrived customer simply leaves. The challenge in the solution is keeping the state of the waiting room synchronized so that when customers or the barber check, they always see a valid state without data races. The key element of a solution is to ensure that only one of the entities can change state of the waiting room at a given time. This is simplified in actor-based solution by maintaining an actor for the waiting room. The implemented solution has an actor for the waiting room, an actor for the barber, and an actor for each customer. Customer actors repeatedly visit the barber shop until they successfully get their haircut. Performance benefits can be gained by scheduling strategies that minimize the number of haircut requests by customers when the waiting room is full. The solution is parametrized by N, the number of customers, and W, the waiting room size. Like the `Dining Philosophers` problem, smarter solutions can minimize the additional round trips made by customers before they get their haircut.

5.5 Cigarette Smokers

The Cigarette Smokers Problem was first presented by Suhas Patil [33], who claimed that it cannot be solved with semaphores. The problem represents a system where applications are waiting on resources and need to be resumed so that they can proceed when resources become available. The problem involves N smokers and one arbiter. A cigarette is made of N ingredients. Each smoker has infinite supply of one unique ingredient. The arbiter has infinite supply of all ingredients. The arbiter repeatedly chooses $N-1$ ingredients at random and puts them on the table, and the smoker who has the complementary ingredient can take the $N-1$ ingredients and make a cigarette to smoke. Meanwhile, the arbiter, seeing the table empty, again chooses $N-1$ ingredients at random and places them on the table. The challenge is to avoid deadlocks by disallowing competing smokers from picking up resources from the table. The actor-based solution maintains an actor for the arbitrator and an actor each for each smoker. Deadlock is avoided as the arbitrator notifies the *eligible* smoker to pick ingredients from the table and all smokers follow this rule. A smoker actor only processes such requests and picks up the ingredients from the table when it is idle (i.e. has completed any smoking activities from previous requests). The solution is parametrized by N, the number of ingredients, and R, the total number of cigarettes smoked successfully before the application terminates.

5.6 Logistic Map Series and Bank Transaction

The synchronous request-response pattern [20, 31] occurs when a requestor sends a request message to a replier system which receives and processes the request, ultimately returning a message in response. With actors, the notion of synchronous request-response occurs when an actor sends another actor a message and stalls further processing of messages until it receives a reply to its message. As the messaging is asynchronous, this pattern requires two messages, a request and a response. While being conceptually elegant, this pattern is hard to implement efficiently, because the requestor actor's single mailbox must handle both the reply message and new messages sent to it from other actors.

We created the Logistic Map benchmark to measure the performance of actor implementations for the synchronous request-response pattern. It computes the Logistic Map [48] using a recurrence relation $x_{n+1} = rx_n(1 - x_n)$. In the benchmark there are three classes of actors: a manager actor, a set of term actors, and a set of ratio actors. The manager actor sends term computation requests to the different term actors and occasionally requests the current term value for a given series. The term actors store the current value of x for a series and are each paired with a unique ratio actor. The ratio actors encapsulate the ratio r and know how to compute the next term x_{n+1} given the current term x_n. The term actor requires a synchronous reply from the ratio actor to update its value of x and, only then, process the next message from the manager to compute the next term in the series. We use non-blocking solutions for all the actor frameworks as thread blocking solutions take much longer time to execute and do not provide a fair comparison. The benchmark is configured by S, the number of series and T, the number of terms to compute per series.

The Bank Transaction benchmark performs transactions between accounts by updating the balances atomically (i.e. the account doesn't process other messages while a transaction is in flight). There are actors to represent each teller and each bank account. The source account first decrements its balance and then needs to synchronously wait for the recipient actor to complete participating in the transaction by incrementing its balance with the same amount. There are three actors involved in a single transaction forming two synchronous request-response chains. The teller is waiting on the source account which in turn is waiting on the recipient account for the transaction to complete. The benchmark is configured by the number of tellers T, the number of account managed by each teller A, and the number of transactions handled by each teller X.

6. Parallelism Benchmarks

In this section we briefly describe the parallelism related benchmarks included in Savina. Parallelism is achieved in actor-oriented programs by executing multiple actors in parallel. The benchmarks include a wide variety of computations that display pipeline parallelism, phased computations, divide-and-conquer style parallelism, master-worker parallelism, and graph and tree navigation.

6.1 All-Pairs Shortest Path

The All-Pairs Shortest Path (APSP) benchmark, represents a phased computation where all actors effectively join on a barrier at the end of each iteration of the outermost loop in Floyd-Warshall's algorithm. In each iteration the slowest actor dominates the execution time of the computation. The computation is implemented by partitioning the matrix to compute into blocks. An actor is created to own each block and this actor is responsible for updating the state of the block in each iteration. An actor is also aware of its neighboring blocks and waits for messages from neighboring actors for block states from the previous iteration (this acts as the barrier). The benchmark is parametrized by N, the number of nodes in the input graph, and B, the size of each block handled by a worker actor.

6.2 A-Star Search

The A-Star algorithm is a popular choice for path finding, because the use of heuristics makes it flexible and allows it to be used in a wide range of contexts. As A-Star traverses a graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way. Thus an efficient solution for A-star requires support for priorities, messages with a higher priority are processed before those with a lower priority even if they were sent earlier. Normally actors do not support multi-level priorities while processing messages which makes this a challenging benchmark to attain performance on. In the `A-Star` benchmark, a randomly generated 3-D grid is searched for a path to a target node from an initial source node. The heuristic of euclidean distance to the goal node is used as the priority to promote processing of nodes closer to the goal node to find the solution. The benchmark is parametrized by W, the number of worker actors; the size of grid, G, where the path search happens; and the number of priority levels, P.

6.3 NQueens first K solutions

In the `NQueens` benchmark, the goal is to find K solutions to placing N queens on a chessboard such that no queen can attack any other. The solution is computed by repeatedly placing a new queen in a non-attacking position on the board until N queens have been placed on the board. The `NQueens` benchmark exploits the ability to support priorities in message processing as well as terminating the computation early once a result is known by ignoring the processing of messages sent earlier. In the `NQueens` benchmark, the goal is to find the first K solutions on a board of size $N \times N$. The divide-and-conquer style is used with master-worker style actors. Each time a worker is able to place a queen on the board, it reports a new work item (along with the state of the board) to the master. The master decides whether the work item is a valid solution or it in turn load balances by forwarding the item to the workers to make further computational progress on the item. The master requests the workers to terminate as soon as the first K solutions are found. To achieve best performance, subproblems deeper in the recursion tree (i.e. items with more queens placed on the board) need to be processed with higher priority by the workers. These priorities help in guiding the execution to processing the request which is likely to yield a solution earlier compared to random processing of messages.

6.4 Recursive Matrix Multiplication

Matrix multiplication is a key linear algebraic kernel, the standard algorithm performs $O(N^3)$ operations while multiplying input matrices A and B of size $N \times N$. `Recursive Matrix Multiplication` benchmark uses a simple recursive divide-and-conquer method for multiplying two dense matrices:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

The divide phase partitions the matrices A and B into size $N/2 \times N/2$ blocks. The conquer phase involves the recursive multiplication of eight $N/2 \times N/2$ matrices and addition of four pairs of results. The conquer phase is parallelized by distributing the work for the eight multiplications among worker actors. The base case of the recursive matrix multiplication uses a simple triply nested loop when the matrix sizes are below a configurable threshold, T. This ensures a fixed number of recursive levels are executed for a given problem size N. The fixed computational structure of the algorithm makes it uniformly balanced; each of the recursive sub-tasks perform equal amount of work and this simplifies the task of load balancing across the worker actors. Care needs to be taken while implementing pairs of multiplication to avoid data races in attempts to update common locations in the result matrix.

6.5 Quicksort

Many algorithms and applications involve interaction patterns that are non-deterministic. `Quicksort` lends itself to divide-and-conquer strategy, however it exposes some amount of non-determinism in availability of partial results. `Quicksort` exhibits both deterministic (creating the left and right fragments around the partition element) and nondeterministic (availability of sorted left and right fragments) forms of task parallelism. Actors can be used to exploit this nondeterminism in the arrival of results from *child* actors as well as guarantee synchronized access while computing partial results. In the actor solution, a tree of actors is formed where a parent actor creates a partition and delegates the work of sorting the left and right partitions to child actors. The child actors report back sorted results to the parent, which in turn merges the result with its currently partially sorted result. The computation completes when the root actor has received and merged results from its child actors. The benchmark is parametrized by N, the size of data to sort and T, the threshold size limit below which to sort the input array fragment sequentially.

6.6 Radix Sort

The `Radix Sort` benchmark has been ported from the SteamIT implementation [43]. Radix Sort is a non-comparison based sort that works by performing successive stable sorts of the next significant radix digit until all digits are sorted. The implementation uses a static pipeline and actor implementations which use message batching to each of the pipeline stages show best performance. All the benchmarks which exploit pipeline parallelism using actors require that messages between the same sender-receiver actor pairs will not be processed by the receiver out of order. The first stage of the pipeline produces a stream of input values and passes it on to the subsequent stages which perform the sort on various bits. During the sort, all entries sorted by the 0-bit value are immediately forwarded to the next stage. The entries sorted by 1-bit value are stored in a temporary buffer and only forwarded after all the inputs have been exhausted. When the values reach the end of the pipeline, the entire input has been sorted. The benchmark is parametrized by N, the size of data to sort and M, the maximum value of the input which determines the number of stages required in the pipeline.

6.7 Filter Bank

The join pattern [3, 40] occurs when messages from two or more data streams are combined together into a single message. Joins have blocking semantics as they need to match inputs from each of the sources and wait until all corresponding inputs become available. The lack of guarantee on which message is processed next, for example by taking the sender of the message into account, makes implementing this pattern troublesome. We use the `Filter Bank` streaming benchmark ported from the StreamIt [43] to quantify the performance of the join pattern. `Filter Bank` is used to perform multi-rate signal processing and consists of multiple pipeline branches. On each branch the pipeline involves multiple stages including multiple delay stages, multiple FIR filter stages, and sampling. Since `Filter Bank` represents a streaming pipeline, it can be implemented using actors. The Branches stage involves a split-join to combine the results of individual Bank stages. Supporting such a join requires maintaining a dictionary to track each sequence arriving from the different banks. The performance of the benchmark is affected by the rate at which the message are delivered across actors and the scheduler that determines which actors are scheduled on the worker threads. The benchmark is configured by N, the number of data items and M, the number of branches to join in the network.

6.8 Bitonic Sort

The `Bitonic Sort` benchmark has also been ported from the SteamIT implementation [43]. `Bitonic Sort` is a divide-and-conquer style comparison-based sorting algorithm. Keys are first ordered into bitonic sequences, one that monotonically increases, then decreases. These sequences are then sorted using a bitonic merger. The actors are arranged in a data-independent sorting network, a sorting network is a mathematical model of a network of wires and comparator modules that is used to sort a sequence of numbers. In the bitonic sort network, the data needs to be received from input sources and sent to output sources in a batched round-robin manner. For example, in one portion of the network an actor may forward the first four messages it receives to actor `A1`, the next four messages to another actor `A2`, the next four messages back to `A1`, and so on. The initial unsorted sequence enters through input pipes, where a series of comparators switch two entries to be in either increasing or decreasing order. The benchmark is configured by N, the number of data items to sort and it must be a power of 2.

6.9 Sieve of Eratosthenes

The `Sieve of Eratosthenes` benchmark uses a pipeline pattern to expose some parallelism. It represents a dynamic pipeline in which a fixed number of local primes are buffered in each stage. Every time the buffer overflows, a new stage (actor) is created and linked to the pipeline, thus growing the pipeline dynamically. There is overhead in filling and draining items in the pipeline for each stage and thus a buffered solution with multiple primes per stage performs better. A stage forwards an item to the next stage only if the item is discovered to be locally prime with respect to the buffered primes in the current stage. Thus, the first few stages of the pipeline process far more elements than the later stages of the pipeline. Hence, some performance benefits can be gained by scheduling actors that represents the first few stages with higher priority. The benchmark is configured by N, the total number of data items to find primes from and B the buffer size at each actor.

6.10 Unbalanced Cobwebbed Tree

The Unbalanced Cobwebbed Tree (UCT) [54] benchmark tries to capture the non-uniform nature of tree exploration computations. It extends the Unbalanced Tree Search benchmark by introducing substantial computations at each tree node, and allowing the computations to be non-uniform. Sizes of the computations follow a normal distribution, with the mean and standard deviation provided as additional parameters during initialization. The benchmark problem is to traverse the computation tree, and carry out the computation at each node when the corresponding node is visited. For extracting good performance from this benchmark, the actor runtime needs to handle dynamic load balancing at a fine granularity by redistributing the available work among the processing units. The benchmark can be configured by multiple parameters such as the maximum number of nodes in the tree and the distribution parameters for the computation size at each node.

6.11 Online Facility Location

Facility Location algorithms are used to decide when and where to open facilities in order to minimize the cost of opening a facility and the cost of servicing customers. In the online version, the locations of customers are not known beforehand and the algorithm needs to make these decisions *on-the-fly*. One algorithm for this problem is the `Online Hierarchical Facility Location` [2]. The algorithm exposes a hierarchical tree structure (quadrants in the algorithm) while performing the computation. Information in the form of customer locations initially flows down the tree. In the algorithm, each node maintains a list of customers it plans to service and this list is partitioned at decision points to form new child nodes. In addition, the decision to create child nodes needs to be propagated up the tree and to selected siblings. A speculatively parallel version of this algorithm can be mapped to use actors. Each actor represents a quadrant which are arranged in a tree structure. Each quadrant maintains a cost value as it receives customers. When the cost threshold is exceeded, the customers are partitioned and transferred to newly formed child quadrants. This represents a dynamic change in the structure of the network by creating new actors to represent the child quadrants. Performance benefits can be gained by introducing parallelism while redistributing the customers to child quadrants. The benchmark can be configured by multiple parameters such as the maximum number of customers, the alpha value that represents the cost threshold, and the grid size of the area being covered by the facilities.

6.12 Trapezoidal Approximation

This `Trapezoidal Approximation` benchmark was initially implemented in Stage [5] and SALSA[46]. It approximates the integral of a function over an interval $[a, b]$ by using the trapezoidal approximation. In the benchmark, we approximate the integral of the function:

$$f(x) = \frac{1}{x+1} \times \sqrt{1 + e^{\sqrt{2x}}} \times \sin(x^3 - 1)$$

The benchmark solves this approximation through standard master-worker style parallelism by dividing the approximation into several intervals. The master actor initiates the computation and creates a set of worker actors. After the different worker actors are started, the work for each interval is statically load balanced across them by a master actor. The worker actors compute the integral approximation in parallel and send their results back to the master actor. The master collects the results from all the worker actors, adds them up, and displays the final result. The benchmark can be configured by the number of workers to spawn and the number of trapezoids to general to compute the integral.

6.13 Precise Pi Computation

`Precise Pi Computation` benchmark computes the value of *PI* to a configured precision in terms of decimal places. The following formula can be used to compute π:

$$\pi = \sum_{n=0}^{\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n$$

This benchmark represents master-worker style actor programs where the master incrementally discovers work to be done and allocates work fragments to the workers. Workers only have at most one message pending in their mailbox and there is no scope for batch processing messages. The master is the central bottleneck in this benchmark as it has to process responses from the workers and send request to compute more terms in the series. The benchmark can be configured by the number of workers to spawn and the number of decimal places accuracy required.

6.14 Successive Over-Relaxation

Iterative methods refer to a wide range of techniques that use successive approximations to obtain more accurate solutions to a linear system at each step. One example of an iterative method is the `Successive Over-Relaxation` (SOR) method. In each iteration, the following formula computed is:

$$T = \frac{D(i-1, j) + D(i, j-1) + D(i+1, j) + D(i, j+1)}{4},$$
$$D(i, j) = \omega D(i, j) + (1 - \omega)T$$

The actor implementation of SOR represents a 4-point stencil computation and was ported from SOTER [45]. Each actor represents an individual cell in the result matrix and the computation mimics multiple iterations of a stencil computation. At each iteration, an actor needs to wait for values to arrive from its neighbors before computing its value for the current iteration and then propagating the values to its dependents. The computation ends when the maximum number of iterations (configurable) has been reached. The benchmark is parametrized by the data size, the maximum number of iterations, and the over-relaxation parameter (ω).

7. Experimental Methodology

The actor libraries used for comparison in this paper all run on the JVM. The libraries are: Akka (AK) [53], Functional Java (FJ) [15], GPars (GP) [41], Habanero-Java Actors (HA) [22–24], Jetlang (JL) [35], Jumi (JU) [13], Lift (LI) [52], Scala actors (SC) [16], and Scalaz (SZ) [21]. All actor implementations of each benchmark use the same algorithm and mainly involved renaming the parent class of the actors to switch from one implementation to the other. All implementations use the pattern matching construct to represent the message processing body (MPB) and hence share the same overheads for the MPB. Similarly, all actor solutions use the same data structures for the user-written code of the benchmarks. We did this to ensure a fair comparison of the internals of the different frameworks.

The benchmarks were run on a 12-core (two hex-cores) 2.8 GHz Intel Westmere SMP node with 48 GB of RAM per node (4 GB per core), running Red Hat Linux (RHEL 6.2). Each core has a 32 kB L1 cache and a 256 kB L2 cache. The software stack includes a Java Hotspot JDK 1.8.0, Akka 2.3.2, FunctionJava 4.1, GPars 1.2.1, Habanero-Java library 0.1.3, Jetlang 0.2.12, Jumi 0.1.196, Lift 2.6-M4, Scala 2.11.0, and Scalaz 7.1.0-M6. We provide a data set configuration for each benchmark in our scripts which can be used to reproduce the results for the benchmarks on different machines. For benchmarking, it is typically desirable to exclude code executed during JVM startup and shutdown from one's measurements. Each benchmark was configured to run using thirteen worker[2] threads and used the same JVM configuration flags (`-Xmx16384m -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:-UseGCOverheadLimit`) and was run for twenty iterations in six separate JVM invocations. The arithmetic mean of the best fifty execution times

[2] one worker thread gets blocked waiting after initialization

(from the hundred and twenty iterations) are reported to minimize effects of JIT and GC overheads from the reported results. In the bar charts, the error bars represent one standard deviation of the fifty execution times. Execution time is measured using JDK's `System.nanoTime()`. We have implemented all of the 30 benchmarks from Table 1 and present results of some of the benchmarks in this section.

7.1 Ping Pong

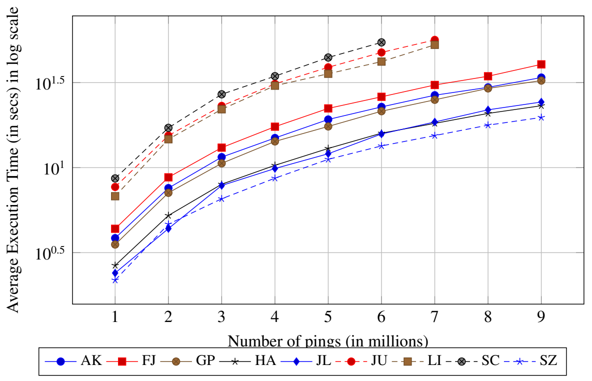

Figure 1: The `PingPong` benchmark exposes the throughput and latency while delivering messages. There is no parallelism to be exploited in the application.

The `Ping Pong` (Figure 1) measures the message passing overheads in the implementation for actors. The SZ, JL, and HA actors perform best. In particular, SZ easily displays the least overhead in messaging. GP, AK, and FJ form the next group. LI, JU, and SC perform the worst of the actor libraries in this benchmark. SC actors are slower due to the use of exceptions to maintain control flow.

7.2 Counting Actor

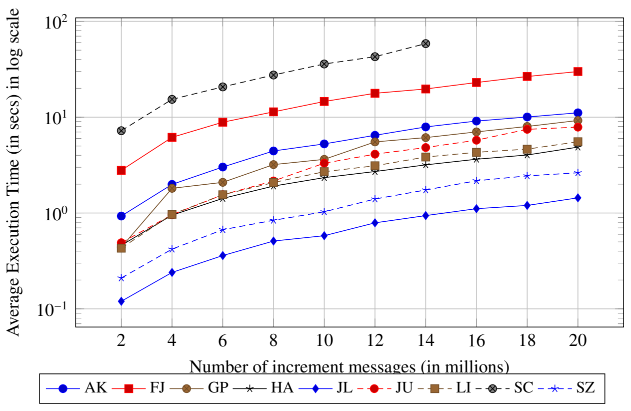

Figure 2: The `Counting` benchmark effectively measures the message delivery in the actor implementation. There is no parallelism to be exploited in the application.

The `Counting Actor` benchmark (Figure 2) measures the message delivery in the actor implementation. In this benchmark, JL performs the best overtaking SZ. JL benefits from the use of batching while processing actor messages instead of creating a new asynchronous task to process each message individually. HA, LI, JU, GP, and AK are bunched close to each other. FJ and SC perform the worst of the actor libraries in this benchmark.

7.3 Fork Join (actor creation) and Fibonacci

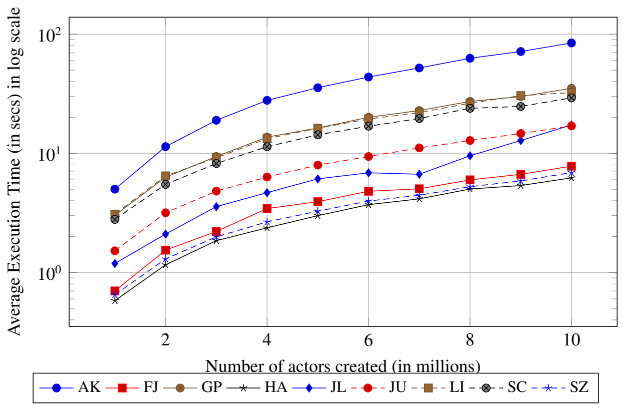

Figure 3: The `Fork Join` (creation) benchmark measures the time taken to create and destroy actor instances.

The `Fork-Join` (creation) and `Fibonacci` benchmarks measure the time taken to create and destroy actor instances. In `Fork-Join` (creation) (Figure 3), all the actor instances are created at the start of the computations and the instances are incrementally destroyed as the computation proceeds. All the libraries perform in a steady manner, HA, SZ, and FJ perform the best while SC, LI, GP, and AK perform the worst. In the `Fibonacci` benchmark (Figure 4), the actors are allocated and deallocated incrementally with most of the actors being short-lived whereas other actors are longer lived. This surprisingly changes the performance of the actor libraries. GP now performs the best as opposed to being amongst the worst performing libraries in the previous benchmark. HA, JL, and FJ show a trend to start performing worse as the fibonacci index value crosses 33. In fact, JL which was initially faster than SZ and JU becomes slower than them as the input size increases.

7.4 Producer-Consumer with Bounded Buffer

In the `Producer-Consumer with a Bounded-Buffer` benchmark (Figure 5), the buffer actor is like the mall actor in the Chameneos benchmark (Section 4.7), so this benchmark also includes measurement of the mailbox contention overhead. The actor-based solutions have the overhead from maintaining additional data structures (adding and removing elements in collections) for the available producers and consumers in the buffer actor. In addition, the benchmark also tests the context switching overheads of the producer and consumer actors which usually process just one message at a time before getting descheduled. SZ and JL perform best benefiting from the efficient concurrent mailbox implementation. AK, HA, and GP have competitive performance

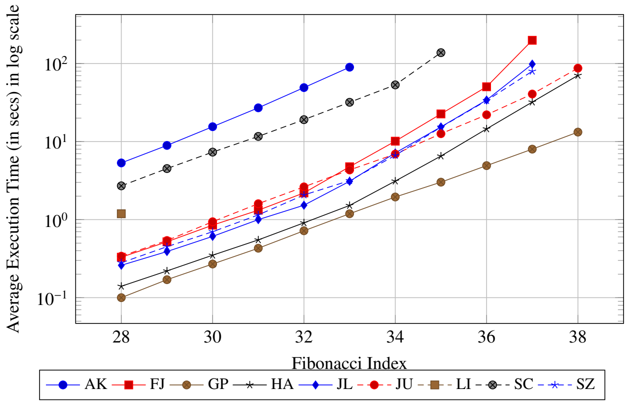

Figure 4: The `Fibonacci` benchmark measures the time taken to create and destroy actor instances.

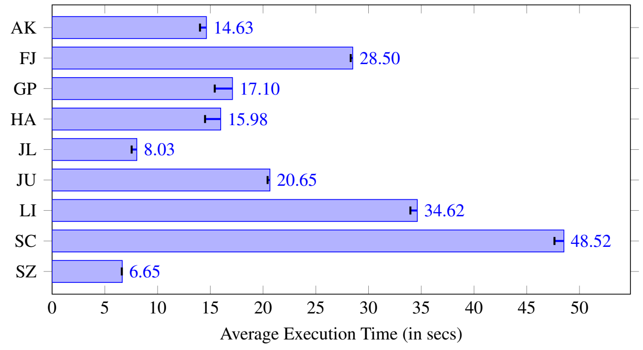

Figure 5: Results of the Bounded Buffer benchmark on bounded buffer size of 6000. There were 5000 producer actors each producing up to 1000 messages. There were 2000 consumer actors.

amongst each other. FJ, LI, and SC perform much slower compared to the other actor implementations.

7.5 Dining Philosophers

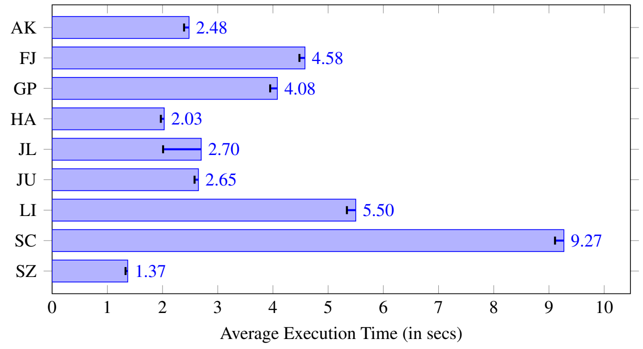

Figure 6: Results of the `Dining Philosophers` benchmark using an arbitrator based solution. Each philosopher attempts to eat 16, 000 times.

The `Dining Philosophers` benchmark also has a central bottleneck in the form of the arbitrator actor. Like the `Chameneos` and `Producer-Consumer with a Bounded-Buffer` benchmarks, SZ performs the best (Figure 6). However, JL is out-performed by HA, AK, and JU. The batch processing of messages ends up hurting JL performance as the same philosopher ends up making futile requests to retrieve forks from the arbitrator. GP, FJ, and LI forms the next group.

7.6 All-Pairs Shortest Path

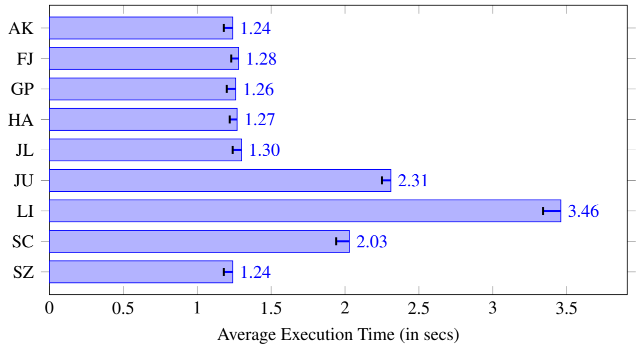

Figure 7: Results of the `All-Pairs Shortest Path` benchmark which uses Floyd-Warshall's algorithm. Average execution time (x-axis) reported in seconds.

The `All-Pairs Shortest Path` benchmark represents a phased computation where all actor effectively join on a barrier in each iteration of the outermost loop in Floyd-Warshall's algorithm before proceeding to the next iteration. In each iteration the slowest actor dominates the computation and as a result we see similar execution times for AK, FJ, GP, HA, JL, and SZ (Figure 7). LI performs surprisingly poorly in this benchmark.

7.7 Bitonic Sort

The `Bitonic Sort` benchmark requires a streaming network as well as round-robin message forwarding. AK performs the best of the actor libraries as shown in Figure 8 due to its improved scheduling of forwarded messages. It is followed by SZ and HA, both benefit from low messaging overhead as well as batch processing of messages at each stage of the network. JL, JU, and LI perform similarly. SC performs the worst.

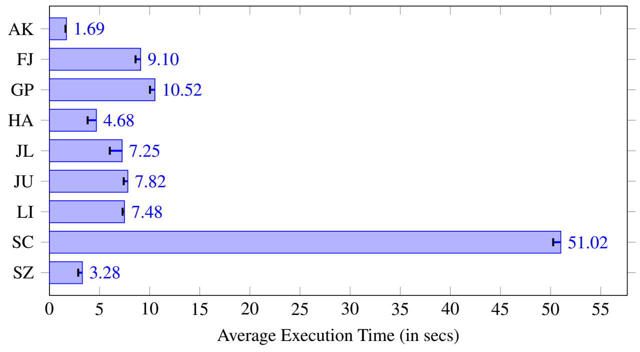

Figure 8: Results of the `Bitonic Sort` benchmark using 32, 768 data items.

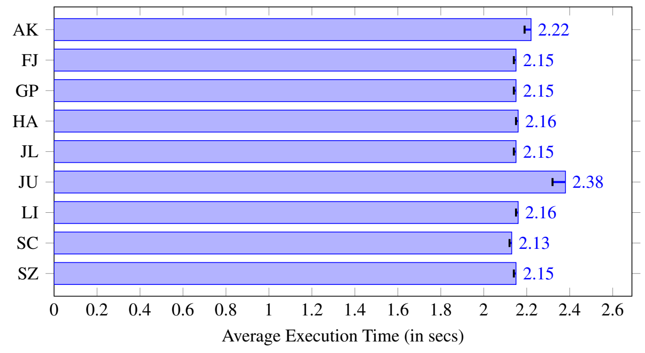

Figure 9: Results of the `Trapezoidal Approximation` benchmark, results displayed for creating 50 million trapezoids.

7.8 Trapezoidal Approximation

In the `Trapezoidal Approximation` benchmark (Figure 9), the master-worker style is used. Since all the work is allocated to the worker actors in advance, most of the actor libraries perform similarly.

8. Summary

We're excited to be introducing the `Savina` benchmark suite for actor-oriented programs. The benchmarks in `Savina` are diverse, realistic, and represent compute intensive applications. We encourage the community to submit open-source solutions to the benchmarks for other actor libraries and languages. This will allow performance comparison across languages and also allow judging the elegance of the solutions.

We plan to add a few more applications into the next version of `Savina`. An important issue we are not addressing with the current release of the suite is inter-language comparisons. Future work will focus on examining a wider range of platforms and environments, and extending the benchmark suite to include codes which use more complex parallel algorithms. We will be making revisions on an ongoing basis in order to fix bugs or expand the scope of the benchmark suite.

Availability

The `Savina` benchmark suites is released open source along with sample inputs and some documentation. The source code for all 30 benchmarks is available online on github at `https://github.com/shamsmahmood/savina`.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF–0964520. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. First and foremost we would like to acknowledge the many authors of the original versions of various benchmark programs which we have been adapted and included in `Savina`. We are very grateful to the anonymous reviewers for their insightful comments and suggestions to clarify the contents of the paper. The paper has been substantially improved based on these suggestions.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[2] A. Anagnostopoulos, R. Bent, E. Upfal, and P. V. Hentenryck. A Simple and Deterministic Competitive Algorithm for Online Facility Location. *Information and Computation*, 194:175–202, November 2004. ISSN 0890-5401.

[3] Apache Software Foundation. Apache Camel: Aggregator, 2004. URL `https://camel.apache.org/aggregator2.html`.

[4] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A Scalability Benchmark Suite for Erlang/OTP. In *Proceedings of Erlang '12*, pages 33–42, New York, USA, 2012. ACM. ISBN 978-1-4503-1575-3.

[5] J. Ayres and S. Eisenbach. Stage: Python with Actors. In *Proceedings of IWMSE '09*, pages 25–32, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3718-4.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of PACT '08*, pages 72–81, New York, USA, 2008. ACM. ISBN 978-1-60558-282-5.

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

[8] R. C. Cardoso, M. R. Zatelli, J. F. Hübner, and R. H. Bordini. Towards Benchmarking Actor- and Agent-based Programming Languages. In *Proceedings of AGERE! '13*, pages 115–126, New York, USA, 2013. ACM. ISBN 978-1-4503-2602-5.

[9] D. Charousset, R. Hiesgen, and T. C. Schmidt. CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications. In *Proc. of the 5th ACM SIGPLAN Conf. on Systems, Programming, and Applications (SPLASH '14), Workshop AGERE!*, New York, NY, USA, October 2014. ACM.

[10] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with "Readers" and "Writers". *Commun. ACM*, 14(10):667–668, October 1971. ISSN 0001-0782.

[11] E. W. Dijkstra. Cooperating Sequential Processes, Technical Report EWD-123. Technical report, Technological University, 1965.

[12] EPCC. The Java Grande Forum Multi-threaded Benchmarks, 2001. URL `http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/s1contents.html`.

[13] Esko Luontola. Jumi Actors. `http://jumi.fi/actors.html`, 2011. URL `http://jumi.fi/actors.html`.

[14] B. Fulgham. The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`, 2009.

[15] functionaljava.org. functionaljava: A Library for Functional Programming in Java, 2010. URL `https://code.google.com/p/functionaljava/`.

[16] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009. ISSN 0304-3975.

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

[18] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Commun. ACM*, 17(10):549–557, Oct. 1974. ISSN 0001-0782.

[19] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, Aug. 1978. ISSN 0001-0782.

[20] G. Hohpe and B. Woolf. Enterprise Integration Patterns - Request-Reply, 2003. URL http://www.eaipatterns.com/RequestReply.html. [Online; accessed 3-April-2014].

[21] L. Hupel and typelevel.org. scalaz: Functional programming for Scala, 2010. URL http://typelevel.org/projects/scalaz/.

[22] S. Imam and V. Sarkar. Integrating Task Parallelism with Actors. In *Proceedings of OOPSLA '12*, pages 753–772, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6.

[23] S. Imam and V. Sarkar. Habanero-Java Library: a Java 8 Framework for Multicore Programming. In *Proceedings of PPPJ'14*. ACM, New York, USA, 2014.

[24] Imam, Shams and Sarkar, Vivek. Habanero-Scala: Async-Finish Programming in Scala. In *Scala Days 2012*, April 2012.

[25] D. Kafura. ACT++: Building a Concurrent C++ with Actors. *Object Oriented Program*, 3:25–37, April 1990. ISSN 0896-8438.

[26] A. Mason. The CountingActor benchmark. http://www.theron-library.com/index.php?t=page&p=countingactor, 2012.

[27] A. Mason. Theron performance. http://www.theron-library.com/index.php?t=page&p=performance, 2012.

[28] A. Mason. The ThreadRing benchmark. http://www.theron-library.com/index.php?t=page&p=threadring, 2012.

[29] R. M. May. Simple mathematical models with very complicated dynamics. *Nature*, 261(5560):459–467, June 1976.

[30] Microsoft Corporation. Asynchronous Agents Library. http://msdn.microsoft.com/en-us/library/dd492627.aspx, 2013.

[31] Oracle. Understanding Interaction Patterns, 2011. URL http://docs.oracle.com/cd/E17904_01/doc.1111/e17363/chapter05.htm.

[32] W. Partain. The nofib Benchmark Suite of Haskell Programs. In J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 195–202. Springer London, 1993. ISBN 978-3-540-19820-8.

[33] S. Patil. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. Technical report, Massachusetts Institute of Technology, 1971.

[34] Rettig, Mike. retlang: Message based concurrency in .NET. http://code.google.com/p/retlang/, 2010.

[35] Rettig, Mike. jetlang: Message based concurrency for Java, 2014. URL http://code.google.com/p/jetlang/.

[36] J. Shirako, N. Vrvilo, E. G. Mercer, and V. Sarkar. Design, Verification and Applications of a New Read-write Lock Algorithm. In *Proceedings of SPAA '12*, pages 48–57, New York, USA, 2012. ACM. ISBN 978-1-4503-1213-4.

[37] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of SPAA '12*, pages 68–70, New York, USA, 2012. ACM. ISBN 978-1-4503-1213-4.

[38] J. Sillito. Stage: Exploring Erlang Style Concurrency in Ruby. In *Proceedings of IWMSE '08*, pages 33–40, New York, USA, 2008. ACM. ISBN 978-1-60558-031-9.

[39] M. Sulzmann, E. S. L. Lam, and P. V. Weert. Actors with Multi-headed Message Receive Patterns. In *Proceedings of COORDINATION'08*, volume 5052, pages 315–330. Springer, 2008. ISBN 3-540-68264-3, 978-3-540-68264-6.

[40] Sybase Inc. Complex Event Processing: Ten Design Patterns. White paper, Sybase - An SAP Company, Apr 2001. URL http://m.sybase.com/files/White_Papers/CEP-10-Design-Patterns-WP.pdf.

[41] The GPars team. The GPars Project - Reference Documentation, 2014. URL http://www.gpars.org/guide/.

[42] The Scala Programming Language. pingpong.scala, 2012. URL http://www.scala-lang.org/node/54.

[43] W. Thies and S. Amarasinghe. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 365–376, New York, USA, 2010. ACM. ISBN 978-1-4503-0178-7.

[44] C. Tismer. Continuations and Stackless Python. In *Proceedings of the 8th International Python Conference*, 2000.

[45] UIUC. SOTER project, 2012. URL http://osl.cs.uiuc.edu/soter/.

[46] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, Dec. 2001. ISSN 0362-1340.

[47] Wikipedia, The Free Encyclopedia. Cigarette smokers problem, 2014. URL http://en.wikipedia.org/wiki/Cigarette_smokers_problem.

[48] Wikipedia, The Free Encyclopedia. Logistic map, 2014. URL http://en.wikipedia.org/wiki/Logistic_map.

[49] Wikipedia, The Free Encyclopedia. ProducerConsumer Problem, 2014. URL http://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem.

[50] Wikipedia, The Free Encyclopedia. ReadersWriters Problem, 2014. URL http://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem.

[51] Wikipedia, The Free Encyclopedia. Sleeping barber problem, 2014. URL http://en.wikipedia.org/wiki/Sleeping_barber_problem.

[52] WorldWide Conferencing, LLC. Lift Framework - LiftActor, 2014. URL http://liftweb.net/api/26/api/#net.liftweb.actor.LiftActor.

[53] D. Wyatt. *Akka Concurrency - Building reliable software in a multi-core world*. Artima Incorporation, USA, 2013.

[54] X. Zhao and N. Jamali. Load Balancing Non-uniform Parallel Computations. In *Proceedings of AGERE! '13*, pages 97–108, New York, USA, 2013. ACM. ISBN 978-1-4503-2602-5.