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 short 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 will be 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. Introduction

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++ (Act++ [16]), Smalltalk (Actalk [5]), Python (Stackless Python [31], Stage [3]), Ruby (Stage [26]), .NET (Microsoft's Asynchronous Agents Library [20], Retlang [22]). Scala brings Erlang style actor based concurrency to the JVM [10]. Since then, many actor libraries and frameworks have been implemented to permit actor-style programming in Java: Jetlang [23], GPars [28], Lift [37], Scalaz [12], Akka [38], Habanero-Java [14], Function-Java [9], 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.

This paper presents Savina, a benchmark suite for actor-oriented programs. 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 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 will be 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 presents our initial experimental results for some of the benchmarks. Section 4 discusses related work and we summarize our conclusions and future work in Section 5.

2. Benchmarks

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 3). 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

#	Name	Symbol	Feature or Pattern being measured	Source
1	Ping Pong	PP	Message delivery overhead	Scala [29]
2	Counting Actor	COUNT	Message passing overhead	Theron [17]
3	Fork Join (throughput)	FJT	Messaging throughput	JGF [6], ourselves
4	Fork Join (actor creation)	FJC	Actor creation and destruction	JGF [6], ourselves
5	Thread Ring	THR	Message sending; Context switching between actors	Theron [19]
6	Chameneos	CHAM	Contention on mailbox; Many-to-one message passing	Haller [11]
7	Big	BIG	Contention on mailbox; Many-to-Many message passing	BenchErl [2]
8	Concurrent Dictionary	CDICT	Reader-Writer concurrency; Constant-time data structure	Ourselves
9	Concurrent Sorted Linked-List	CSLL	Reader-Writer concurrency; Linear-time data structure	Shirako et al. [24]
10	Producer-Consumer with Bounded Buffer	PCBB	Multiple message patterns based on Join calculus	Sulzmann et al. [27]
11	Dining Philosophers	PHIL	Inter-process communication; Resource allocation	Wikipedia [34]
12	Sleeping Barber	SBAR	Inter-process communication; State synchronization	Wikipedia [36]
13	Cigarette Smokers	CIG	Inter-process communication; Deadlock prevention	Wikipedia [33]
14	Logistic Map Series	LOGM	Synchronous Request-Response with non-interfering transactions	Ourselves ([35])
15	Bank Transaction	BTX	Synchronous Request-Response with interfering transactions	Ourselves
16	Radix Sort	RSORT	Static Pipeline; Message batching	StreamIT [30]
17	Filter Bank	FBANK	Static Pipeline; Split-Join Pattern	StreamIT [30]
18	Sieve of Eratosthenes	SIEVE	Dynamic Pipeline	GPars [28]
19	Unbalanced Cobwebbed Tree	UCT	Non-uniform load; Tree exploration	Zhao and Jamali [39]
20	Online Facility Location	OFL	Dynamic Tree generation and navigation	Ourselves
21	Trapezoidal Approximation	TRAPR	Master-Worker; Static load-balancing	Stage [3]
22	Precise Pi Computation	PIPREC	Master-Worker; Dynamic load-balancing	Ourselves
23	Recursive Matrix Multiplication	RMM	Uniform load; Divide-and-conquer style parallelism	Ourselves
24	Quicksort	QSORT	Non-uniform load; Divide-and-conquer style parallelism	Ourselves
25	All-Pairs Shortest Path	APSP	Phased computation; Graph exploration	Ourselves
26	Successive Over-Relaxation	SOR	4-point stencil computation	SOTER [32]
27	A-Star Search	ASTAR	Message priority; Graph exploration	Ourselves
28	NQueens first *N* solutions	NQN	Message priority; Divide-and-conquer style parallelism	Ourselves

Table 1: List of `Savina` Benchmarks divided into three categories: 7 micro-benchmarks, 8 concurrency benchmarks and 13 parallelism benchmarks.

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 will be 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 others 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 7 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 13 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 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 will 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.

3. Experimental Results

The actor libraries used for comparison in this paper all run on the JVM. The libraries are: Akka (AK) [38], Functional Java (FJ) [9], GPars (GP) [28], Habanero-Java Actors (HA) [13, 15], Jetlang (JL) [23], Jumi (JU) [7], Lift (LI) [37], Scala actors (SC) [10], and Scalaz (SZ) [12]. 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, Functional-Java 4.1, GPars 1.2.1, Habanero-Scala 1.0, 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[1] 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 (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 19 of the 28 benchmarks from Table 1 and present results of three benchmarks below.

[1] one worker thread gets blocked waiting after initialization

3.1 Mailbox Contention (Chameneos)

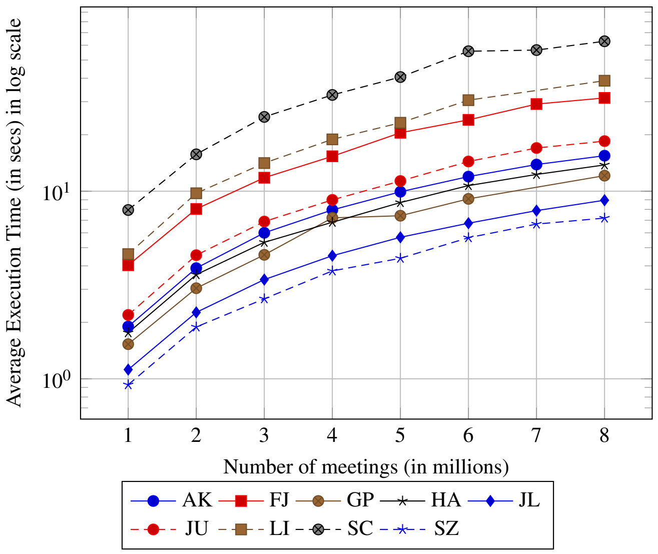

Figure 1: The Chameneos benchmark was run with 500 chameneos (actors) constantly *arriving* at a mall (another actor). There are multiples of millions meetings between chameneos orchestrated at the mall.

The `Chameneos` micro-benchmark, shown in Figure 1, measures the effects of contention on shared resources while processing messages. The original SC implementation was obtained from the public Scala SVN repository [11]. The benchmark involves all *chameneos* constantly sending messages to a mall actor that coordinates which two *chameneos* get to meet. Adding messages into the mall actor's mailbox serves as a contention point and stress tests the concurrent mailbox implementation. The SZ version performs best with JL following closely. The next set of GP, HA, AK, and JU actors are slightly slower, but competitive. The SC version pays the penalty of generating exceptions to maintain control flow in its `react` construct. In general, all the implementations scale linearly with an increase in the input size.

3.2 Logistic Map Benchmark

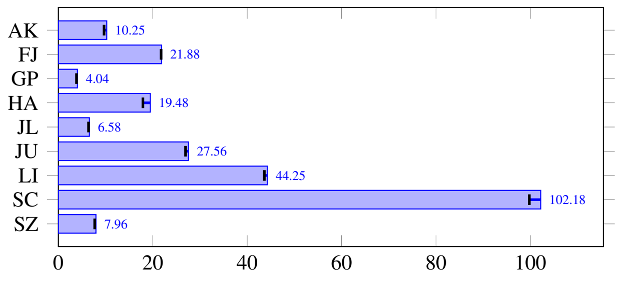

Figure 2: Results of the LogisticMap benchmark using 150 term actors and 150 ratio actors. Each term actor is responsible for computing 150000 terms. Average execution time (x-axis) reported in seconds.

We created the Logistic Map benchmark to measure the performance of actor implementations for the synchronous request-response pattern. It computes the Logistic Map [35] 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 ratio actors encapsulate the ratio r and know how to compute the next term given the current term x_n. The term actors require a synchronous reply from the ratio actor before they update their value of x and, only then, process the next message from the master to compute the next term in the series. We use non-blocking solutions for all the actors frameworks, thread blocking solutions take much longer time to execute and do not provide a fair comparison as some solutions might be non-blocking. Our solution for the AK version uses a custom extension that allows individual unstashing of messages, by default the Akka library only allows `unstashAll` which introduces a lot more overhead. Figure 2 displays the results of this benchmark for the various actor implementations. GP, JL, SZ, and AK perform the best with HA, FJ, JU, and LI following in the next set. SC performs noticeably poorly.

3.3 Split-Join Benchmark (Filter Bank)

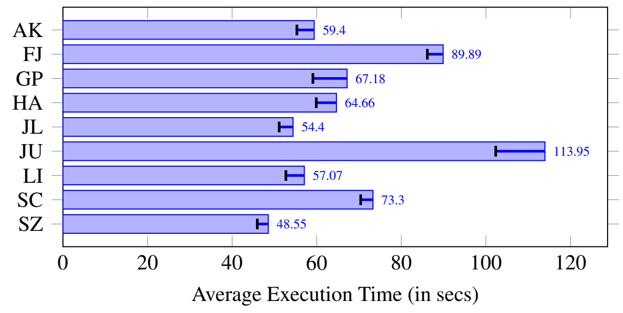

Figure 3: Results of the Filter Bank benchmark results configured to use 8-way join branches. The input used 300, 000 data items and 131, 072 columns. Average execution time (x-axis) reported in seconds.

We use the Filter Bank streaming benchmark ported from the StreamIt [30] 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. Figure 3 compares the performance of the Filter Bank benchmark against the different actor library implementations. SZ, JL, LI, and AK perform best followed by HA and GP. JU performs noticeably poorly.

4. 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 [2]. 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 micro-benchmarks [18] with detailed performance analysis.

`PARSEC` [4] 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 [25]. 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 [21] 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 [8] captures a broad set of languages, it compares over 20 programming languages on a set of 13 micro-benchmarks.

5. 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.

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] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A Scalability Benchmark Suite for Erlang/OTP. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, Erlang '12, pages 33–42, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1575-3. doi: 10.1145/2364489.2364495. URL `http://doi.acm.org/10.1145/2364489.2364495`.

[3] J. Ayres and S. Eisenbach. Stage: Python with Actors. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, pages 25–32, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3718-4. doi: http://dx.doi.org/10.1109/IWMSE.2009.5071380. URL `http://dx.doi.org/10.1109/IWMSE.2009.5071380`.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New

York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454128. URL http://doi.acm.org/10.1145/1454115.1454128.

[5] J.-P. Briot. *Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment*, pages 109–129. Cambridge University Press, 1989. URL http://web.yl.is.u-tokyo.ac.jp/members/briot/actalk/papers/actalk-ecoop89.ps.Z.

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

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

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

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

[10] 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. Distributed Computing Techniques.

[11] 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.

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

[13] S. Imam and V. Sarkar. Integrating Task Parallelism with Actors. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 753–772, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6.

[14] S. Imam and V. Sarkar. Habanero-Java Library: a Java 8 Framework for Multicore Programming. In *11th International Conference on the Principles and Practice of Programming on the Java Platform*, PPPJ'14. ACM, New York, NY, USA, 2014.

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

[16] D. Kafura. ACT++: Building a Concurrent C++ with Actors. *J. Object Oriented Program*, 3:25–37, April 1990. ISSN 0896-8438. URL http://dl.acm.org/citation.cfm?id=90482.90493.

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

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

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

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

[21] 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. doi: 10.1007/978-1-4471-3215-8_17. URL http://dx.doi.org/10.1007/978-1-4471-3215-8_17.

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

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

[24] J. Shirako, N. Vrvilo, E. G. Mercer, and V. Sarkar. Design, Verification and Applications of a New Read-write Lock Algorithm. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 48–57, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4. doi: 10.1145/2312005.2312015. URL http://doi.acm.org/10.1145/2312005.2312015.

[25] 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 the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4. doi: 10.1145/2312005.2312018. URL http://doi.acm.org/10.1145/2312005.2312018.

[26] J. Sillito. Stage: Exploring Erlang Style Concurrency in Ruby. In *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, pages 33–40, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-031-9. doi: http://doi.acm.org/10.1145/1370082.1370092. URL http://doi.acm.org/10.1145/1370082.1370092.

[27] M. Sulzmann, E. S. L. Lam, and P. V. Weert. Actors with Multi-headed Message Receive Patterns. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, volume 5052 of *COORDINATION'08*, pages 315–330. Springer, 2008. ISBN 3-540-68264-3, 978-3-540-68264-6.

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

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

[30] 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*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: 10.1145/1854273.1854319. URL http://doi.acm.org/10.1145/1854273.1854319.

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

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

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

[34] Wikipedia, The Free Encyclopedia. Dining philosophers problem, 2014. URL http://en.wikipedia.org/wiki/Dining_philosophers_problem.

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

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

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

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

[39] X. Zhao and N. Jamali. Load Balancing Non-uniform Parallel Computations. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '13, pages 97–108, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2602-5. doi: 10.1145/2541329.2541337. URL http://doi.acm.org/10.1145/2541329.2541337.