Survey on intra-actor Parallelism in the Actor Model

Michael Rüfenacht
m.ruefenacht@students.unibe.ch
Joint Master in Computer Science
Workshop 2013
University of Neuchâtel, Switzerland

***Abstract*—The paradigm shift from centralized and serial to distributed and concurrent computing, introduced the need for models that fulfill the inherently concurrent nature of such systems. One example of a model, which recently regained researcher's attention, is the Actor Model. Concurrency issues in thread-based systems are explicitly handled by the application layer what appears to be complex and error prone. The Actor Model, in comparison, implicitly avoids possibly arising concurrency issues such as race conditions. Its strict semantics introduce strong guarantees, but also limit the scalability of actor-based systems. One of the Actor Model's main drawbacks is the difficulty to exploit intra-actor parallelism when processing incoming messages. This paper presents a survey of the model itself and approaches targeted at enriching the Actor Model with parallel processing of messages and the concurrent execution of code sections within actors.**

I. Introduction

Created by Carl Hewitt et al. [1] in the early seventies, the Actor Model was further formalized and theoretically refined in publications from Grief [2] Hewitt et al. [3] or Agha [4] just to name a few. The widespread use of multi-core architectures and distributed computing introduced the need for adequate models to safely exploit concurrency [5]. The Actor Model fits the characteristics and requirements of parallel and concurrent systems by design and inherently avoids arising concurrency issues. The model and its entities underly strict data access and behavioral semantics. In contrast to thread-based environments, concurrency in the Actor Model is achieved by using asynchronous message passing.
Beside academic interests and the possibility of reasoning on its internal state, the Actor Model meanwhile was implemented in a number of languages[1] and integrated with frameworks, such as the Akka library[2]. Usual concurrency problems such as race conditions, deadlocks, starvation or liveness issues cannot occur in actor-based systems. Despite being beneficial in many aspects the aforementioned strong semantics constrain concurrency and therefore limit scalability. Attempts to exploit the full benefits of parallelism to improve scalability, rely on relaxations of the of the Actor Model and its semantics. To increase the level of abstraction, the approaches reviewed in the following sections, introduce appropriate mechanisms such as additional language constructs e.g. for convenient handling of synchronization. This survey presents two approaches which target at the improvement of scalability of the basic Actor Model by enabling parallel message processing.

Outline. The paper starts with an overview of the main Actor Model characteristics in Section II. Section III explores the concept of Parallel Actor Monitors (PAM) which allow the injection of message schedulers into actors. In Section IV, I discuss the Unified Model (UM) which combines the Actor Model with the Async-Finish Model (AFM). Section V briefly compares PAM to the UM and their capabilities to exploit intra-actor parallelism. Section VI concludes before I briefly discuss the effectiveness of the presented concepts.

II. The Actor Model

In essence, the Actor Model consists of two concepts: the actor entities and asynchronous message passing. Actors ([1]) are self-contained ([6]), strongly encapsulating entities. They can be composed to systems and communicate by asynchronous message passing [6] under strong semantics as formalized by Agha [7], Clinger [8] or Grief [2].

A. The actor

Actors as the main entities of the Actor Model follow the *'everything is an actor'* credo, similar to the *'everything is an object'* paradigm. Actors have a restricted set of fundamental capabilities:

- send a finite amount of messages to other actors
- receive a finite amount of messages from other actors

[1]such as Erlang, Scala, AmbientTalk …
[2]http://akka.io/

- spawn new actors
- change their internal state (become another actor)

One of actor's most important characteristics are also the Actor Model's strongest limitations: strong encapsulation and locality. An actor can be stateful but the Actor Model removes the feasibility to access another actor's internal state. Although an actor can expose information about its state by appropriate messages, there exists no shared mutable state. In general, actors can be viewed as black-boxes, having references to other actors instances only. This concept of strong encapsulation inherently creates mutual exclusion without synchronization and locking. As a consequence, an actor is constrained to send messages to already known addresses from incoming messages or spawned actors only, which is referred to as locality.

B. Asynchronous message passing

Communication between actors is handled by asynchronous message passing. A message consists of an actor's address (or identifier), the return address and the data (sometimes referred to as *message payload*). Depending on the Actor Model implementation, an actor determines appropriate behavior based on the type of the incoming message.

Actors store received messages in an unique storage facility known as a mailbox. The Actor Model ensures fairness by providing weak message delivery guarantees, meaning that every message gets eventually delivered. In Object-Oriented Programming the invocation of a method on an object can lock the invoking entity. The emission of messages in the Actor Model happens asynchronously and does not block the sender. The non blocking communication enables the actors to work in parallel and avoids locking of system resources [9] (e.g. a system stack).

An actor is not required to process messages in any particular order (what Hewitt et al. refer to as *particular order of events* [3]). The absence of constraints on the order of message delivery corresponds to non-determinism, an essential property of parallel computing. Nevertheless, an actor is only permitted to process one message at a time, a decision that was made upon the introduction of stateful actors [10]. Allowing two messages to be processed in parallel leads to concurrent access to the internal state of the actors. While this limitation guarantees the absence of deadlocks and data races, it puts a serious limitation on task level parallelism. Also, the absence of shared mutable state requires libraries implementing the Actor Model to support call-by-value semantics [9] (i.e. which would avoid shared references).

C. Parallel message processing and data partitioning

As elaborated earlier, an actor is only allowed to process one message at a time which implies serial processing of the internal message queue.

Nevertheless it is possible to establish a certain degree of parallelism by delegating parts of the computation to other actors, which would process them in parallel. An actor can therefore implement a *divide-and-conquer* strategy and spawn new actors that compute partial results on a chunk of data. Scholliers et al. [11] emphasize the resulting coupling between data and computation.
One strategy to enable the delegation of data-coupled tasks to new actors, is data partitioning [12]. An actor therefore is responsible to create data chunks (of incoming or contained data) and for provisioning them together with the necessary parameters to the spawned actors upon initialization. Since the Actor Model does not permit the parent actor and the spawned actors to have access to the same data, the parent actor has to release its lock on the partition or create a copy before passing it to the child. In this case Actor Model semantics can be fully preserved, but partitioning and copying of the data are expensive tasks and cause the majority of the mentioned overheads. Furthermore, executing global actions on the distributed data chunks are accompanied with a high degree of overhead associated with controlling and coordination. Examples of global actions are search operations over several actors or data manipulations, that require other actors to update their part of the data.

D. Improvements

The Actor Model introduces several desirable characteristics one example being its abstraction of concurrency and strong guarantees for concurrent systems. As stated previously, these advantages introduce a trade-off between data-race-freedom and efficiency caused by the sequential message processing inside an actor. Attempts such as data partitioning to achieve the desired task level parallelism without violating the Actor Model, cause overheads [11]. The following approaches try to establish other strategies to improve parallelism (and therefore introduce violations of the model and its semantics).

E. An example of message processing

To demonstrate the differences and benefits the following approaches provide, we consider a contrived table actor. It stores an encapsulated, two dimensional data structure (a table), organized in indexed rows which are accessible via ID. The table actor acts as a read/write service for an arbitrary number of clients (similar to an indexed table in a relational database management system). The restriction to process one message at a time enforces the actor to subsequently execute both, the read and the write messages, under mutual exclusion. The resulting impossibility to exploit parallelism in the given scenario creates a bottleneck and hence limits scalability. Figure 1, provides an example of a message arrival sequence, comparing computation time (x-axis) to the progress in message queue processing (y-axis).

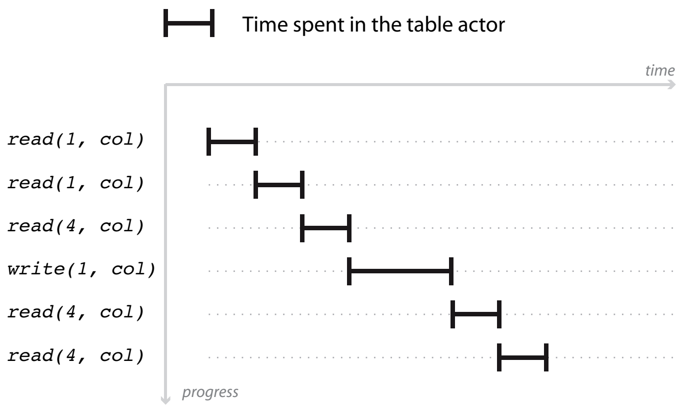

Fig. 1: default serial execution of read and write accesses to different rows (and columns)

Implementing a data partition strategy in this context is straightforward. The table actor spawns row actors and passes the corresponding data chunks to them. Hence, reads and writes to disjoint rows can be executed in parallel (the table actor still only processes one message at the time, by simply delegating the access), illustrated in Figure 2. The emerging overheads depend on the complexity and size of the underlying data structure. In addition, the partitioned processing causes a larger memory footprint and complicates global actions. A good example for such an operation would be searching, which would require consulting all rows (probably even locking them) and handle the join of the results (an issue that was also identified by Imam et al. [13] and solved in their Unified Model implementation). Additional attention should be paid to the type of the underlying computation. Benefits associated with executing partial computations depend on how much inherent parallelism a given task can exhibit (i.e., Amdahl's Law [14], [15]).

III. Parallel Actor Monitors (PAM)

Scholliers et al. introduced Parallel Actor Monitors (PAM) [11]. PAM are schedulers that inject themselves into actors for exploiting potential parallelism, which could be obtained in a thread-based environment. PAM were developed to establish parallelism without causing the overheads associated with data partitioning elaborated in subsection II-C. PAM enable messages to be processed

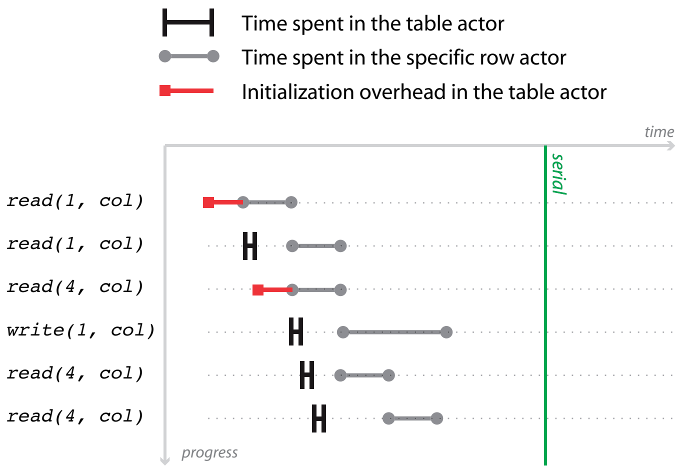

Fig. 2: partially parallel execution after data partitioning

in parallel by a threadpool, using a scheduling policy, which depends on an actor-specific strategy.

A. *The parallel actor monitor*

Scholliers et al. describe PAM as schedulers that implement an actor-specific coordination strategy for the execution of incoming messages. PAM therefore permit messages to be safely processed in parallel by the underlying thread pool. Scheduling strategies vary from basic, *single-writer/multiple-readers* strategies, up to highly complex protocols [11]. The injection of PAM is similar to the strategy design pattern, used in object-oriented programming, and has four main characteristics:

- **Efficiency:** Speedup gained by parallel computation (where it is possible)
- **Modularity:** PAM are reusable and parameterizable components that are injected into actors and reside between the actor's message processing code and the message queue. An important aspect of PAM is that their usage does no imply refactoring of existing code i.e. the message-processing body [11].
- **Locality:** The scheduling strategy of a given actor monitor is local and not visible to other actors.
- **Abstraction:** The parallel actor monitor is an abstract entity as the actors themselves and is responsible for the handling of the messages in the queue. Thread access is handled by the runtime and is completely transparent to the application layer.

The procedure PAM perform can be illustrated best by stepping through an example of a message processing workflow. The `schedule` action is is the method triggering the scheduling of messages while `leave(msg)` is a callback that indicates the termination of a running thread processing the passed message. Both actions synchronized and are executed under mutual exclusion as visualized in Figure 3.

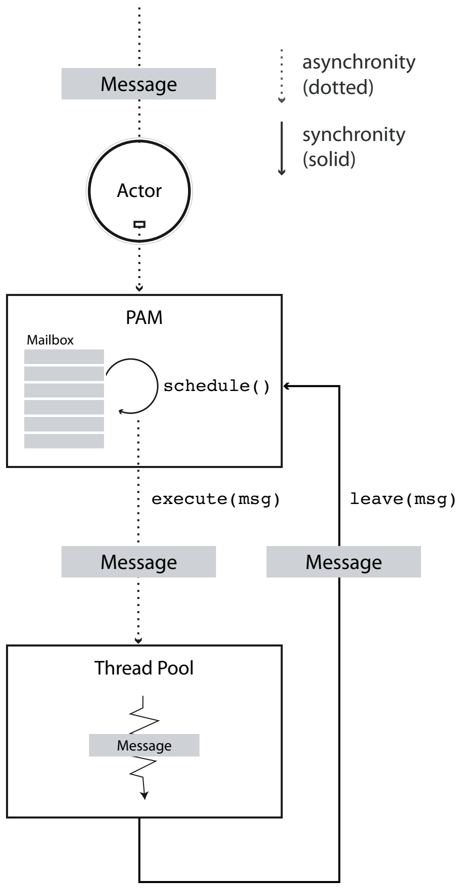

Fig. 3: The workflow of a PAM as proposed by Scholliers et al. [11]

1) Incoming messages are handed over to PAM.
2) PAM queue messages in a mailbox until the scheduling procedure runs.
3) The messages matching the strategy are scheduled for execution. They are assigned to a thread out of the thread pool of the runtime. While the messages are processed, the code executed in the thread has free access to the actor's state.
4) After completion of the execution, the thread which executed that task runs the `leave` callback method of the monitor and PAM schedules remaining messages (if the queue is not empty).

B. Consequences and application

On one hand, the injection of PAM leaves the initial actor-specific code untouched, what makes it modular and reusable. On the other hand, the workflow unveils a relaxation of the model: the encapsulation of an actor and its thread environment is broken during the execution of the message. This reintroduces possible concurrency issues such as data races that need to be explicitly avoided by the implemented scheduling policy. During scheduling however, known techniques for identifying errorneous behavior or a flaw in the implemented scheduling strategy, such as Datarace Detection [16], [17] were not used. Figure 4 shows the positive impact of the exploited thread-parallelism with a multiple readers, single writer scheduler. The read accesses can be immediately executed in parallel reducing the total execution time. The write message is processed under mutual exclusion, depicted in the red area. As long as we are accessing only two different rows the performance is similar to the data partitioned actor in Figure 2 indicated by the brown line. As soon as the actor is required to access more disjoint rows the overhead of data partitioning would grow. If the strategy could distinguish between the accessed rows, the reads after the write stage could also happen in parallel since they access a different row.

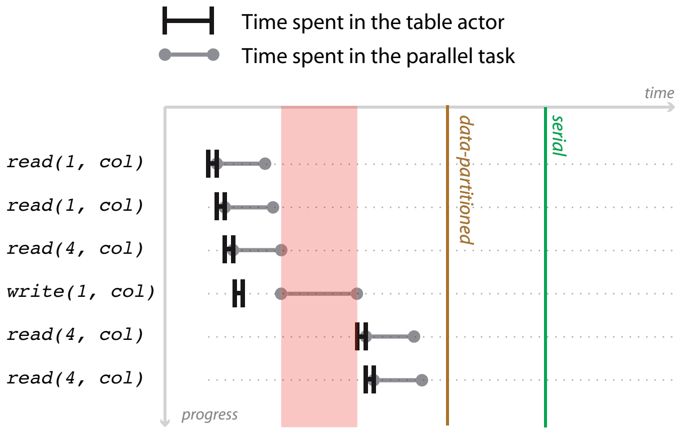

Fig. 4: The example processing powered by a PAM where the red area indicates the mutual exclusion. All the read accesses happen in parallel.

IV. The Unified Model

An alternative approach for exploiting parallelism was proposed by Imam and Sarkar [13]. The authors integrated the divide-and-conquer strategy of the Async-Finish Model (AFM) into actor-based environments, namely Scala and Habanero Java. The combination of the AFM and the AM is called the Unified Model (UM). Furthermore, they introduced several enhancements to the AFM itself, for the purpose of circumventing some of its identified limitations.

A. *The Async Finish Model (AFM)*

The Async-Finish Model allows the creation (fork) and the joining of lightweight parallel tasks. To facilitate language integration, the authors provide corresponding language constructs, named `async` and `finish`. Tasks created in an `async` block can be wrapped into a common `finish` scope (join), having arbitrary levels of nesting. All tasks are enclosed into an unique dynamic Immediately Enclosing Finish (IEF). An IEF ensures that all forked processes terminate until program execution proceeds. The IEF can be explicitly created or is implicitly given for the entire program. Similar to PAM, the provided thread allocation is handled by the runtime. The Async-Finish Model works best if used with deterministic algorithms, ensures determinism in the absence of data races but is unable to handle non-determinism [13]. Imam et al. state Quicksort as an example where the AFM is not able to exploit the non-deterministic availability of the sorted fragments and i.e. do computations on early available results.

B. *Data-Driven Futures*

The proposed AFM implementations permit the usage of Data Driven Futures (DDFs) in combination with an additional language construct, called `asyncAwait`. A DDF is construct served by a producer (thread that does the computation) and awaited by a consumer (thread that awaits the result of the computation) and consists of three main callbacks:

- `put(value)` is invoked upon successful completion of the computation by the producer. Once a value is put into the future, the object gets hardened (becomes immutable) and the waiting consumer gets notified by the runtime.
- `await()` is invoked by the `asyncAwait` and the consumer is delayed by the runtime until a value was put into the DDF by the producer.
- `get()` can be invoked by the consumer after awaiting the DDF. The previous waiting guarantees there was a put before and the DDF is associated by a value.

A parent task is capable of asynchronously awaiting the termination of an arbitrary number of subtasks (each represented through a DDF) without blocking the program execution. The combination of the async-finish constructs with DDFs introduces the possibility for parallelization inside the message-processing body (MBP) [13]. DDFs are a suitable replacement for constructs like count-down latches (which represent shared state) to model join behavior. The presence of a blocking, enclosing finishing scope (multiple scopes are executed serially), prevents data races as well as avoids parallel execution of messages.

C. *Scope escaping and the paused state*

As stated by Imam et al. [13], the enclosure of the processing of a message into a single finishing scope leads to serial execution of messages. To circumvent this limitation, the authors implemented the functionality to escape an actors own finishing scope. Escaping permits an actor to return immediately to the initial control flow without awaiting the termination of the spawned tasks inside the scope. The possibility to escape a scope is thought to be used only if the message does not require the internal state to be mutated. Nevertheless, the constraint of not mutating the actors state is neither monitored nor ensured, what enables potential data races.
To ensure the mutually-exclusive processing of critical sections emerging if multiple tasks run in parallel, an actor is able to be paused (by a `pause` method). When an actor in paused state, it stops the processing of messages in the mailbox. To resume the actor from a paused state, the awaited task is required to invoke the `resume` method explicitly upon termination.

D. *Consequences and application*

On one hand, the capacity to escape the finishing scope enables the Unified Model to spawn tasks that process messages in parallel. In addition to being able to escape the finishing scope of an actor it is possible to exploit parallelism within the spawned task (intra-task parallelism). Critical sections (tasks that modify the actors state) can be protected by pausing the actor what prevents him from spawning new, concurrent tasks. On the other hand, escaping the scope reintroduces shared access to the actor's state and possible data races, which violates then fundamental guarantees provided by the Actor Model. As elaborated previously also for PAM, the occurrence of data races is not monitored or reported. Imam et al. indicated that they plan to investigate the integration of Dynamic Datarace Detection for future work. Dynamic Datarace Detection would enable the runtime to react to data races and e.g. reschedule erroneous messages. Figure 5 demonstrates the behavior of the previously introduced table actor. Figure 6 provides an insight on the ways DDFs can be used to read from multiple rows.

Even the rudimentary example in Figure 5 suffers from possible data races. In reality the example does not fully exhibit intra-task parallelism and processing would be the same as in Figure 4. Assuming we could establish intra-task parallelism in the write operation, we would arrive at a timeline, depicted in Figure 7.

```
def behavior() = {
        // single writer
        case Write(row, column) => {
                finish {
                        async {
                                // an independent action
                        }
                        async {
                                // write into the row
                        }
                } // block untill all asyncs finish
        }
        // multiple readers
        case Read(row, column) => {
                async {
                        // read the specified column
                }
                // escape scope immediately
                // proceed with behavior while the
                // reading runs
        }
}
```

Fig. 5: Skeleton of a reader-writer implementation in the UM in Scala. The behavior method is dedicated to implement the message handling of the actor.

```
Continued from Figure 5...
// searches in two disjoint rows in parallel
case Search(value) => {
        // result DDFs
        val (result1, result2) = (ddf(), ddf())
        async {
                search(value, result1)
        }
        async {
                search(value, result2)
        }
        asyncAwait(result1, result2) {
                // get the awaited results from the
                // DDFs and combine them
        }
}
```

Fig. 6: The skeleton of a fork-join search in a table actor, implemented exploiting the benefits of the Unified Model in Scala. Spawned tasks (async) are not explicitly embraced in a finishing scope, providing the possibility to execute searches in parallel.

V. Comparison

Consider a brief overview of the characteristics of the AM, PAM and the UM, provided in Table I. The Actor Model is able to establish improved parallel processing of messages by data partitioning, suffering from data copying overheads. Parallel Actor Monitors as well as the Unified Model enable Actor Model implementations to process messages in parallel and improve scalability (as stated in a number of benchmarks [11], [13]) at the cost of data races and a certain loss of abstraction. While thread allocation and coordination is handled transparently for

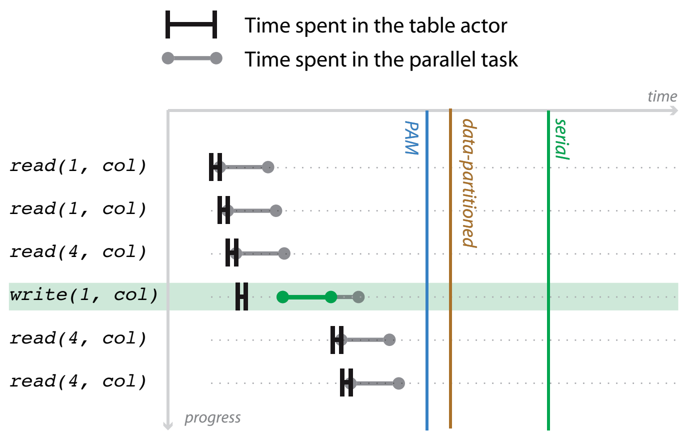

Fig. 7: An example processing sequence where we assume that the write stage benefits from intra-task parallelism and is executed faster.

the programmer, the proposed mechanisms neither ensure the absence of data races, nor provide the capacity to detect data races. The concrete implementation of both approaches has to explicitly handle arising concurrency issues what can lead to additional complexity. PAM offer the capabilities to exploit enhanced parallelism without altering already existing Actor source code. The UM, on the other hand, requires rewriting existing Actor-specific code, but provides convenient mechanisms to implement fork-join functionality and exploit intra-task parallelism.

TABLE I: Comparison of the approaches

	AM	PAM	UM
task parallelism	●	✓	✓
intra-task parallelism	●	×	✓
Advantages	data race free abstract	modular	fork-join parallelism
Disadvantages	scalability	no encapsulation guarantee	data races complexity

✓supportet, ● partially supported, × not supported

VI. Conclusion

While both, PAM and the UM, exploit intra-actor parallelism and solve their respective objectives, they reintroduce possible data races. The handling, as well as detection and avoidance of data races has to be managed at a low level of the concrete implementation, which contradicts with the abstraction the Actor Model offers. As both approaches are relatively recent, further improvements on the presented approaches or different ways of exploiting intra-actor parallelism are very likely to be developed. One example could be the implementation of reliable mechanisms such as Dynamic Datarace Detection, for monitoring data races.

Acknowledgments

I'd like to thank Yarco Hayduk for his patience and the frequent reviews of my work.

References

[1] Carl Hewitt, Peter Bishop, and Richard Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proceedings of the 3rd international Joint Conference on Artificial Intelligence*, August 1973.

[2] Irene Greif, "Semantics of communicating Parallel Processes," Cambridge, MA, USA, Tech. Rep., 1975.

[3] C. Hewitt and H. Baker, "Laws for communicating parallel processes," *Information Processing 77*, pp. 987–992, 1977.

[4] Gul Agha, "Actors: A Model Of Concurrent Computation In Distributed Systems," Ph.D. dissertation, 1985.

[5] Herb Sutter and James Larus, "Software and the Concurrency Revolution," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.

[6] Gul Agha, "Concurrent Object-Oriented Programming," *Commun. ACM*, vol. 33, no. 9, pp. 125–141, September 1990.

[7] Gul Agha, "The Structure and Semantics of Actor Languages," in *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, 1991, pp. 1–59.

[8] William Douglas Clinger, "Foundations of Actor Semantics," Cambridge, MA, USA, Tech. Rep., 1981.

[9] Rajesh K. Karmani, Amin Shali, and Gul Agha, "Actor Frameworks for the JVM Platform: a comparative Analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09. New York, NY, USA: ACM, 2009, pp. 11–20.

[10] Akinori Yonezawa, Ed., *ABCL: An Object-Oriented Concurrent System.* Cambridge, MA, USA: MIT Press, 1990.

[11] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter, "Parallel Actor Monitors: Disentangling Task-Level Parallelism from Data Partitioning in the Actor Model." in *14th Brazilian Symposium on Programming Languages*, 2010.

[12] Roxana Diaconescu, "Object based Concurrency for data parallel Applications: Programmability and Effectiveness," Ph.D. dissertation, Norwegian University of Science and Technology, 2002.

[13] Shams Imam and Vivek Sarkar, "Integrating Task Parallelism with Actors," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '12, New York, NY, USA, 2012, pp. 753–772.

[14] Gene Myron Amdahl, "Validity of the single Processor Approach to achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485.

[15] Guihai Chen. Transcript of Amdahl's original talk (with notes). [Online]. Available: http://inst.eecs.berkeley.edu/~n252/sp07/Papers/Amdahl.pdf

[16] Guang-Ien Cheng, "Algorithms for Data-Race Detection in Multithreaded Programs," 1998.

[17] Jong-Deok Choid, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan, "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs," *SIGPLAN Not.*, vol. 37, no. 5, pp. 258–269, May 2002.