High-Throughput Stream Processing with Actors

Luca Rinaldi, Massimo Torquati, Gabriele Mencagli, Marco Danelutto
luca.rinaldi,torquati,mencagli,marcod@di.unipi.it
Computer Science Department, University of Pisa, Italy

Abstract

The steady growth of data volume produced as *continuous streams* makes paramount the development of software capable of providing timely results to the users. The Actor Model (AM) offers a high-level of abstraction suited for developing scalable message-passing applications. It allows the application developer to focus on the application logic moving the burden of implementing fast and reliable inter-Actors message-exchange to the implementation framework.

In this paper, by using the CAF framework as reference AM implementation, we focus on evaluating the model in high data rate streaming applications targeting scale-up servers. Our approach leverages Parallel Pattern (PP) abstractions to model streaming computations and introduces optimizations that otherwise could be challenging to implement without violating the Actor Model's semantics. The experimental analysis demonstrates that the new *implementation skeletons* we propose for our PPs can bring significant performance boosts (more than 2×) in high data rate streaming applications.

Keywords: Actor Model, Parallel Patterns, Data Stream Processing, Programming Model, Multi-Cores

ACM Reference Format:
Luca Rinaldi, Massimo Torquati, Gabriele Mencagli, Marco Danelutto. 2020. High-Throughput Stream Processing with Actors. In *Proceedings of (AGERE! 2020).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The ever-increasing volume of data produced in the form of *continuous streams*, has made scalable concurrency a fundamental aspect of the software development process.

Actor-based languages and frameworks are more and more employed to design and develop complex streaming applications that need high flexibility, adaptivity, and high-scalability [13]. The Actor Model (AM) of computation [6, 26, 47] offers a high-level of abstraction that allows developers to focus on their application business logic while the underlying implementation framework takes care of implementing fast, memory-safe, and reliable messaging systems.

However, in the AM, the scalability concept is often associated with scale-out settings (i.e. large distributed systems or clusters). This is due to its share-nothing pure message-passing model, which somehow trades single node performance for scalability to a large number of nodes [16]. Nonetheless, solutions capable of consolidating several distributed servers in a single scale-up multi-core have recently gained special attention since they can reduce HW costs, software licenses cost, data-center space, and power consumption [11]. Numerous recent research efforts in the direction of designing Stream Processing Engines (SPEs) bear with this trend [38, 39, 48]. To this end, the "pure" AM does not offer significant margins for enhancing its efficiency on a single scale-up multi-core. This is primarily due to the impossibility of exposing the physical shared-memory to Actors, and to introduce low-level optimizations in the messaging systems without breaking the semantics of the model [44].

In our previous works [42, 43], to face the performance optimization issues of the AM on a scale-up server, we proposed to enhance the model with a set of well-known *Parallel Patterns* (PPs). PPs are integrated into the AM as "macro Actors". The low-level platform-specific optimizations and the exploitation of the shared-memory are confined within the *implementation skeleton* of the PPs and they are entirely transparent to the application programmer. The programmer has the responsibility to select the proper PP to solve his/her problem. In contrast, the patterns provider has the responsibility to produce an efficient and memory-safe implementation of PPs for the target platform (*separation of concerns* software design principle). This approach trades the full flexibility of the AM with increased efficiency on the single node.

Problem statement. In contrast to the *batch* processing model, the *continuous streaming model* processes input data as soon as it is available. The aim is to minimize results latency while keeping up with the input rate. This processing model is adopted in several streaming applications whose typical software architecture is exemplified in Fig. 1.

The application processes continuous streams of data from a set of sources (e.g., IoT sensors, financial markets). The input data are then processed and aggregated through a

AGERE! 2020, ,
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

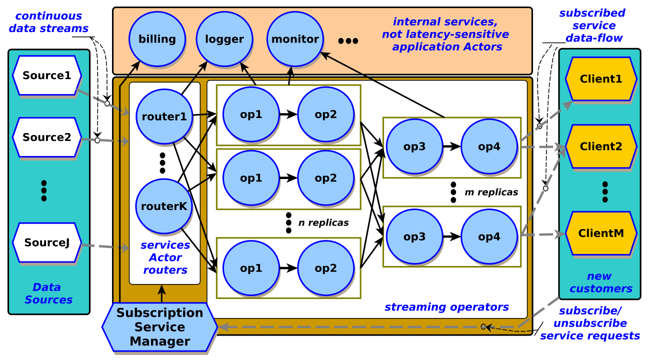

Fig. 1. Typical software schema of streaming application.

network of Actors (typically having a DAG topology), each one implementing a *stateless* or *stateful* operator (e.g., *filter*, *reducer*, *flat-map*). Customers (which are the sinks of the network) dynamically subscribe/unsubscribe application services, which *de facto* are continuous streams of output results (e.g., in the financial domain, chart patterns within stock prices). The application graph also comprises a set of non-latency critical Actors implementing specific features (e.g., logging, billing, disk data recording). Depending on the input data rate, the number of sources, and the number of services offered, single operators may be replicated several times, paying attention to carefully routing messages to the correct next Actor in case of stateful operators.

These streaming applications could be conveniently implemented by using Actor Model implementations (e.g., CAF [21], Akka [25]), because of the many concurrent entities involved, the explicit management of message routing, and the lack of shared states. Actually, in the stream analytics domain, specialized SPEs model applications that have static data-flow graphs of operators executed by dedicated threads [2]. Our objective is to understand whether a more powerful and flexible model of computation (i.e., the AM), is suitable for the execution of high data rate stream analytics applications. We want to investigate if it is possible to preserve the expressive and flexibility power of the AM and to execute high-throughput streaming operators capable of coexisting with standard Actors.

To assess the AM performance figures in this application context, we simulated a simplified version of the schema in Fig. 1 by using the C++ Actor Framework (CAF) [21, 22]. Precisely, we implemented a *linear pipeline* of three Actors: *Source*, *Forwarder* and *Sink* (cf. Sec. 4). We observed that the maximum rate our microbenchmark can sustain is more than 2 times lower than the one obtained by implementing the same benchmark "manually" with native C++ threads. Besides, the average message latency is about 3 times higher.

Careful performance analysis has shown two issues:

1. The messaging system's complexity for managing different message types in an Actor is a limiting performance factor in Actor-based streaming applications where streaming operators usually deal with a single data type[1].
2. At high input rates, the unlimited capacity of Actors' mailboxes pushes too much pressure in the memory system, making it challenging to stabilize the application behavior and limit memory consumption.

Contributions. In this work, we tackle the two issues discussed above. We utilized CAF as a reference implementation of the AM. Specifically, our contributions are:

- we show the performance limits of Actor-based implementations in high-throughput data stream processing computations;
- we propose a new *implementation skeleton* of the *Sequential* PP (the basic building-block of our PPs), which has been specifically optimized for high-throughput and low-latency data streaming computations in scale-up servers;
- we propose a new *implementation skeleton* of the *Farm* PP that optimizes *Pipeline* compositions of consecutive *Farm* PPs to reduce the number of message hops;
- we discuss the issues of unbounded mailboxes in streaming computations. To introduce a basic backpressure mechanism for streaming operators, we propose a new communication primitive that takes into account the queue length of the receiving Actor.

Using a set of well-known streaming benchmarks, we demonstrate the performance advantages of using the new implementations in the C++ Actor Framework.

Outline. Sec. 2 provides some background information. Sec. 3 introduces the Parallel Pattern Actors used to enhance the performance of Actor-based programs on scale-up servers. Sec. 4 proposes a new *implementation skeleton* of some Parallel Patterns targeting high-throughput streaming computations. Sec. 5 presents the experimental evaluation, with the considered applications and the results in terms of throughput. Sec. 6 discusses some related works, and Sec. 7 summarizes the results.

2 Background

The Actor Model. The Actor Model (AM) [6, 7, 33] is a well-established approach to concurrent and distributed computations. It addresses the challenges of *data races* in concurrent shared-memory programming with threads by forbidding shared mutable state among Actors. Actors exchange immutable messages, and the only mechanism for observing or modifying internal Actor states is to implement a message-based protocol. Each Actor has a private *mailbox* of unbounded capacity, which stores input messages. The processing of messages is performed asynchronously and atomically, and there is no guarantee on the processing order.

[1]It is worthwhile mentioning the sources of overhead we faced are not related to potential implementation flaws of the CAF library

The *memory isolation*, the *message-passing style of communications*, and the *serial processing of messages* ensure data-race freedom in Actor-based programs.

Pattern-based parallel programming. One of the well-established approaches for raising the level of abstraction in parallel computing is based on the concepts of *Parallel Patterns* (PPs) [37], which are customizable schemes of parallel computations that recur in many applications and algorithms [27]. These parallel abstractions are made available to programmers as high-level programming constructs with a well-defined functional and extra-functional semantics. Each parallel pattern has one or more implementation schema (called *implementation skeleton*) for a given target platform [24]. Notable examples of PPs are *Pipeline*, *Map-Reduce*, *Task-Farm*, *Divide&Conquer*. PPs are used in several programming frameworks and libraries such as Microsoft PPL [20], Intel TBB [41], SkePU [30] and FastFlow [9].

Data Stream Processing. Several broadly used Stream Processing Engines (SPEs), such as Storm [2] and Flink [1], adopt the *continuous streaming model* where inputs are processed as soon as they are available, and output results are produced continuously to provide timely information to the users. Streams are *unbounded* sequences of data coming from one or more sources. A source produces data *tuples* of the same type. Distinct sources may produce tuples of different data types. Streaming applications are modeled as DAGs of *stateless* or *stateful operators* that produce results based on their business logic. Operators can be replicated to keep up with high input data rates as well as to increase the system throughput. To deal with streams, many SPEs provide operators that repeatedly apply the processing on a *window* of recent tuples. This is enabled by the so-called *sliding window* processing model [32], where a window is a bounded set of the most recent tuples whose content is dynamically determined according to various semantics (e.g., count-based, time-based).

The CAF library. The C++ Actor Framework (CAF) is an Actor-based framework implemented in modern C++ [21, 22]. The framework follows the *Classical Actor Model* [26] initially proposed by Agha [6]. Actors define a set of *behaviors*, each activated upon the receipt of a specific input message type. CAF implements *behaviors* through C++ lambda functions. The appropriate lambda function will be selected through a *pattern matching* between its formal parameters and the input message types. Moreover, CAF and all implementation of the *Classic Actor Model*, enables run-time change of Actor behaviors employing the *became* primitive.

By using the *send* primitive Actors can send any sequence of data types into the mailbox of other Actors. Such types sequence will be moved to a *type-erased tuple* that erases the actual type preserving annotations of the erased types. These annotations will be used in the pattern matching phase to discover the behavior to execute and to cast back the types to the original ones. Besides, the CAF messaging system supports two priority levels and the possibility to skip unwanted messages (e.g. messages for behaviors not yet set up). CAF Actors uses a combination of a LIFO lock-free queue with multiple FIFO buffers to implement the mailbox. The LIFO queue is a thread-safe unbounded linked-list with an atomic pointer to its head. There is one FIFO linked-list for each priority level. CAF supports two priority levels. Each FIFO queue also has an additional cached buffer to maintain messages that are skipped. The sender Actor inserts new elements atomically in the head of the LIFO queue. The receiver Actor, atomically extracts all the messages from the LIFO queue using a *compare-and-swap* operation. Then, the messages are divided into the two FIFO queues on the basis of their priority. Finally, the consumer Actor can dequeue messages with a different proportion from the two queues to maintain the priority.

CAF allows multiple Actors to implicitly share message contents, as long as no Actor performs writes. This permits sending the same content to multiple Actors without any copying overhead provided the message handler of receiving Actors take an immutable reference (*copy-on-write*).

A Work-Stealing algorithm is used to dynamically assign ready Actors to run-time threads. However, it is also possible to assign private threads to Actors by spawning *detached* and *blocking* Actors. They are useful for implementing particular functionalities or executing non-blocking I/O operations.

3 Parallel Pattern Actors

Recently we proposed to use Parallel Pattern abstractions to enhance the performance of the Actor Model in shared-memory systems [42, 43]. The motivation is twofold: a) to introduce a communication structure in Actor-based applications that usually are characterized by unstructured communication topologies [31], and b) to safely enable some low-level shared-memory based optimizations that are generally not allowed by the "pure" Actor Model[2].

The set of PPs we defined, is sketched in Fig. 2. We provided *Data-parallel PPs* namely *Map* and *Divide&Conquer* (D&C); and *Control-parallel PPs* namely *SeqActor*, *Pipeline*, and *Farm*. Fig. 3 shows an example of a possible composition and nesting of PPs in Actor-based networks.

In our previous works, we focused mainly on enabling shared-memory exploitation to speed up *data-parallel computations* such those present in some well-known multi-core benchmarks (e.g., PARSEC [18]). These computations, usually work on independent partitions of large arrays or matrices through *parallel-for* and *Map-Reduce*. Therefore, we implemented the *Map*, and also the *D&C* Parallel Pattern

[2]The implementation, called *CAF-PP*, is available at https://github.com/ParaGroup/caf-pp.

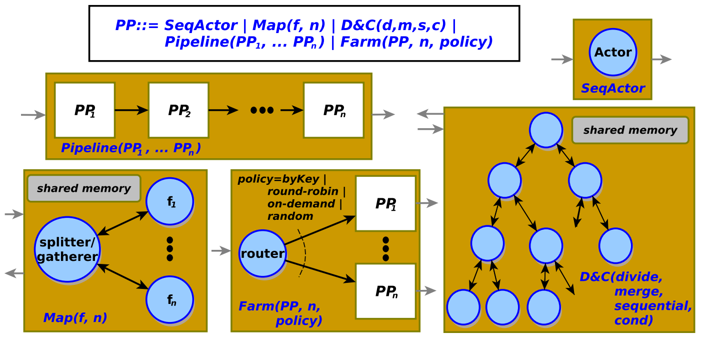
PP::= SeqActor | Map(f, n) | D&C(d,m,s,c) | Pipeline(PP1, … PPn) | Farm(PP, n, policy)
Actor
SeqActor
PP1
PP2
PPn
Pipeline(PP1 , … PPn)
shared memory
f1
fn
splitter/gatherer
Map(f, n)
policy=byKey | round-robin | on-demand | random
router
Farm(PP, n, policy)
D&C(divide, merge, sequential, cond)

Fig. 2. The Parallel Pattern Actors (PPAs).

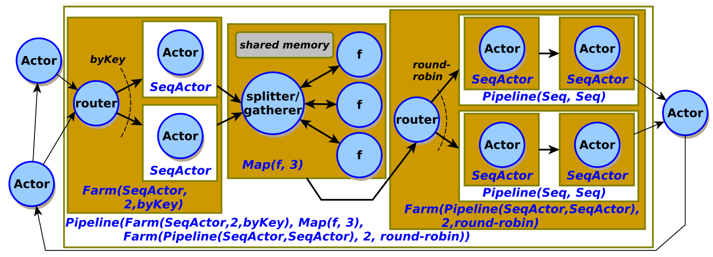
Actor
byKey
router
SeqActor
Farm(SeqActor, 2,byKey)
shared memory
splitter/gatherer
f
Map(f, 3)
round-robin
Pipeline(Seq, Seq)
Farm(Pipeline(SeqActor,SeqActor), 2,round-robin)
Pipeline(Farm(SeqActor,2,byKey), Map(f, 3), Farm(Pipeline(SeqActor,SeqActor), 2, round-robin))

Fig. 3. Example of composition and nesting of PPAs.

using the shared memory within their *implementation skeletons*, avoiding data copy, and enhancing the performance of fine-grained synchronizations.

This approach using PPs for speeding up data-parallel computations allowed us to obtain performance close to specialized thread-based multi-core libraries such as FastFlow [9], providing at the same time the memory-safe environment of the Actor Model to application programmers.

Data-flow PPs such as *SeqActor*, *Pipeline*, and *Farm* have been initially introduced mainly to enable patterns composition, nesting and to provide the programmer with well-defined structured topologies in the AM. The *SeqActor* pattern is an Actor wrapper. It is used to integrate standard CAF Actors within PP Actors (e.g., for creating a pipeline of Actors). The *Pipeline* implements a parallel composition of multiple Parallel Pattern working in parallel on subsequent data elements. It takes care of connecting each Parallel Pattern in the correct order. The *Farm* pattern models Parallel Pattern replication. Each distinct PP replica (usually called *Worker*), works in parallel on distinct data elements of the input stream. Stream elements are forwarded to the Workers according to some predefined scheduling policy (e.g., round-robin, random, byKey), or by using a user-defined policy (through a C++ lambda function). If the number of replicas is left unspecified, a default value will be used. The most important feature of these patterns is their ability to compose different PPs in a functional way. *Sequential*, *Map* and *D&C* cannot contain nested patterns, and they must be used as leaves of the data-flow tree representing the application (or part of it) implemented with Parallel Patterns.

In this work, our objective is to enhance the performance of *Control-parallel* patterns, focusing on streaming applications and their ever increasing demand of high-throughput and low end-to-end message latency. Specifically, we propose a different *implementation skeleton* for the *Sequential* and *Farm* patterns that optimizes the message exchange in the composition of PPs within Actor-based applications.

4 Streaming Parallel Patterns Actors

The Actor Model is in principle particularly suited to implement streaming applications because of the many concurrent entities usually involved, the explicit management of messages routing, and the absence of shared states among operators. An Actor can be mapped one-to-one with a streaming operator; thus, the Actor programmer may concentrate on implementing the operators' business logic without worrying about how inter-Actors communications happen.

In the context of high-throughput demanding streaming applications targeting scale-up servers, the use of system-level languages for implementing the AM is unquestionably a performance plus compared to, for example, Java-based implementations The C++ Actor Framework (CAF) allows the development of Actor-based programs leveraging modern C++. Differently from other well-known implementations of the Actor Model, such as Erlang [12] and Akka [10], which use virtual machine abstractions, CAF applications are compiled directly into native machine code. However, CAF does not provide specific support for data-intensive streaming applications[3].

To evaluate the performance of CAF Actors in streaming computations, we implemented two microbenchmarks.

Microbenchmarks. By using the Parallel Pattern *Pipeline*, we connected three *SeqActors* in a linear chain. The first Actor (called *Source*) generates a stream of tuples at maximum speed (each tuple is 24 B). The second Actor (called *Forwarder*) forwards each input tuple to the next stage, and finally, the last Actor (called *Sink*) collects all tuples. Then, the same pipeline has been implemented by using three C++ threads and two FIFO lock-free queue [8] to implement the channels between the stages. The two implementations were executed for 60 s and their throughput was measured in the Sink node. CAF (version `0.17.5`)[4] has been configured to run with three run-time Worker threads and with the *aggressive* polling strategy that maximizes system reactivity [45]. Fig. 4 shows the results obtained. The measured throughput by employing CAF Actors is more than 2 times lower than the thread-based implementation (1.4 *vs* 3.3 Mtuples/s). To better understand the problem, we used a second microbenchmark implementing a simple Producer-Consumer

[3]CAF offers experimental support for data-flow streams between Actors [3]. We did not use such a building-block in our implementation because, from the performance standpoint, it does not solve the issues outlined in Sec. 4
[4]We also tested the pre-release `0.18` obtaining similar performance figures.

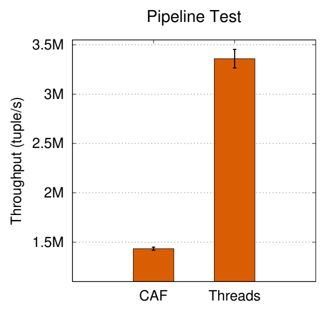

Fig. 4. Three stage pipeline microbenchmark. CAF-based *vs* "manual" thread-based implementation.

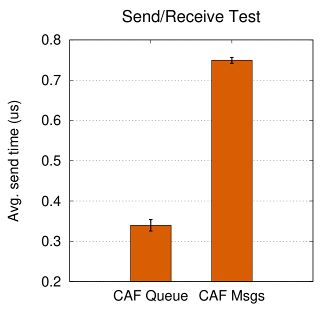

Fig. 5. Prod-Cons. microbenchmark. Simplified (CAF Queue) *vs* default CAF messaging system.

pattern using two CAF Actors (P and C) exchanging 100 M small messages (4 B each). The objective is to evaluate the overhead for exchanging a single message between P and C. The first version uses the lock-free queue used in the CAF run-time for implementing Actors' mailboxes as a communication channel between P and C (cf. Sec. 2). Therefore, P pushes the messages directly into the queue while C pops them out. In the second version, instead, we used the default CAF messaging system (which leverages the same lock-free queue with the dynamic message dispatching). Fig. 5 shows the results of this second microbenchmark. The overhead introduced by the Actors messaging system is more than 2 times the base case (0.33 µs *vs* 0.75 µs). Such overhead is due to the complexity of managing different types of messages, even if, as in our microbenchmarks, the Actors will always receive the same input type for the entire execution. This extra cost, perhaps, can be lowered by fine-tuning the implementation code, but certainly, it cannot be completely removed without loosing the flexibility of defining multiple behaviors for an Actor.

In streaming applications, operators work on statically defined input and output data types (the type of input and output tuples), and the extra complexity of managing multiple data types is the primary source of overhead.

Streaming Operators. To tackle the problem that appeared in the microbenchmarks, we propose a new *implementation skeleton* for the *Sequential* pattern capable of handling a single message type and optimized for defining high-throughput data streaming operators in CAF. Such new skeleton is implemented by the *SeqNode* class whose interface is inspired to the one provided by the *node* building-block in the FastFlow parallel library [9]. Fig. 6 shows its current interface. Each new *operator* must define at least the consume method, which is called as soon as a tuple is available to be consumed by the node. During the *SeqNode* execution, input tuples are processed sequentially and in order. The other two methods on_start and on_stop, are automatically invoked once when the node starts and right before it terminates, respectively. These virtual methods may be overwritten in the user-defined operator to implement initialization and finalization code for the operator node.

The *SeqNode* has a typed input queue and zero, one or more typed output queue references (implemented by the multiQueues object in Fig. 6 line 15) in order to connect the operator implemented by the node to one or more different *SeqNodes*.

```
 1 template<typename Tin, typename Tout=Tin>
 2 class SeqNode {
 3 public:
 4   void run() {
 5     // execute the node
 6   }
 7 protected:
 8   virtual void consume(Tin & x) = 0;
 9   virtual void on_start() { /* nop */ }
10   virtual void on_stop()  { /* nop */ }
11   void send_next(Tout && x) {
12     // ...
13   }
14   Queue<Tin> in_;
15   std::optional<multiQueues<Tout>> out_;
16 };
```

Fig. 6. The base interface of a streaming operator.

```
struct Operator: SeqNode<T> {
  Operator(MyState s): SeqNode{}, s_{s} {}
  void consume(T &x) override {
    T y = do_something(x, s_);
    send_next(std::move(y));
  }
private:
  MyState s_;  // operator local state
};

int main() {
  //...
  MyState s1, s2;
  Operator op1{s1};
  Operator op2{s2};
  Pipeline pipe{op1, op2};

  auto p= spawn_pattern(sys,pipe).value();
  //...
}
```

Fig. 7. Simple example showing how to define an operator node and how to use it in a *Pipeline* pattern.

Therefore, the *Sequential* pattern has two implementation skeletons: *SeqActor* and *SeqNode*. The main differences between them is that the first behaves like a standard CAF Actor and so it can exchange messages with any other Actor as it is a standard CAF Actor or a PP. On the contrary, the *SeqNode* can be used only inside a *Pipeline* pattern or it can be a Worker of a *Farm* pattern. The *Pipeline* and *Farm* patterns provide the necessary interface to enable the *SeqNode* to communicate with other standard Actors. Fig. 7 shows how to define *SeqNode* operators and how to connect

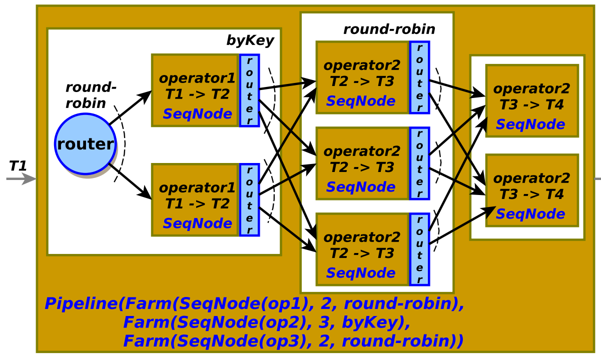
byKey
round-robin
round-robin
router
T1
T4
operator1 T1 -> T2 SeqNode
operator2 T2 -> T3 SeqNode
operator2 T3 -> T4 SeqNode
router
Pipeline(Farm(SeqNode(op1), 2, round-robin),
Farm(SeqNode(op2), 3, byKey),
Farm(SeqNode(op3), 2, round-robin))

Fig. 8. Optimized *Pipeline* composition of three *Farm*s patterns running replicas of *SeqNode* operators.

them in a *Pipeline* PP. *SeqNode*s are currently implemented as CAF *blocking* Actors. The message exchange between two consecutive *SeqNode*s does not rely on the CAF messaging system. Instead, typed messages are pushed directly into the input queue of the receiving node. As shown in the microbenchmark tests, this approach permits to greatly reduce the message-exchange overhead present in standard Actors, related to the complexity of managing pattern-matching for the dynamic message dispatching.

Pipeline compositions of Farms. The functional-style composition is one of the primary features of the proposed PPs. Streaming applications can be easily modeled by one or more *Pipeline* compositions of *SeqNode*s where some operators are replicated using the *Farm* pattern and suitable distribution policies. The *Farm* implementation skeleton uses a router Actor to implement the pre-defined or user-defined distribution policies towards *Farm*'s Workers (see Fig. 2). The router Actor introduces an extra message hop for each operator replica in a streaming network formed by multiple *Farm* compositions. Clearly, these extra hops may have an impact on the end-to-end latency for traversing the entire network of operators and may introduce a bottleneck in case of high data rates and fine-grained operators. For these reasons, we decided to introduce a new implementation skeleton for the composition of *Farm* PPs within a *Pipeline* to reduce the number of router Actors hence of message hops.

When a *Pipeline* pattern connects two consecutive *Farms* (regardless the kind of PP compositions used in the Workers), the router Actor of the second farm is automatically removed and its distribution policy is implemented within the *send_next* method of the right-most leaf PPs (i.e., *Sequential*, *Map*, and *D&C*) present in all Worker replicas of the first *Farm*. Fig. 8 shows a simple use-case in which three *Farm* patterns are used within a *Pipeline* to replicate three *Sequential* operators (in the figure we used *SeqNodes*). Only the first *Farm* preserves the router Actor because it can receive messages from outside the pattern, e.g., from a standard CAF Actor. On the contrary, the second and third *Farm*(s) do not have the router Actor.

The `send_next_if` primitive. In almost all SPEs is implemented a form of backpressure to guarantee that sender Actors cannot overload receiver Actors due to different relative speed [36]. Instead of implementing complex and costly demand signaling protocols between each sender and receiver, we decided to implement a more straightforward form of flow-control by providing a `send_next_if` command within Parallel Patterns. The *if* condition is applied to the actual number of messages present in the destination queue. If the length of the queue (observed by the sender) is greater than the specified parameter value, the send command immediately returns to allow the user to take actions (e.g., waiting a while before retrying or discarding the message or buffering it locally). The `send_next_if` command, like `send_next`, uses the policy configured by the next pipeline stage to select the queue in which to insert the message. However, if the next stage policy is either *round-robin* or *random*, the command will try to enqueue the messages in all next queues before returning with failure. This simple mechanism allows the sender to implement flow-control strategies and to auto-regulate the speed of sending messages. In high data rate scenarios, such control-flow strategies are typically needed only in the *Source* operator.

We modified the CAF LIFO queue by adding a new method that returns the estimated length of the queue (i.e., `synchronized_size`). This method will be internally used by the `send_next_if` command to check the queue length of *SeqActors* and *SeqNode*s. `synchronized_size` counts the elements present both in the multiple FIFO queues and in the LIFO thread-safe queue. In the first case the count is performed without synchronization and might return an approximated value, instead the LIFO queue count is performed within a spin-lock to avoid data-races. However, in the implementation we made, the extra cost of the synchronization is payed only in the `send_next_if` where the queue length is needed and not in the `send_next`.

5 Evaluation

The experiments were conducted on a Intel Xeon multi-core equipped with a dual-socket Intel E5-2695 Ivy Bridge CPUs running at 2.40GHz and featuring 24 cores (12 per socket 2-way hyper-threading). Each hyper-threaded core has 32 KB private L1, 256 KB private L2 and 30 MB of L3 shared cache. The server has 48 logical cores, 64 GB of DDR3 RAM. It runs Linux 4.15.0 x86_64 with the CPUfreq performance governor enabled. We used gcc `9.0.1` with the `-O3` optimization flag. The CAF version is `0.17.5`, and for all tests, the number of run-time Worker threads was set to the total number of Actors used. The Work-Stealing CAF run-time has been configured to use the most *aggressive* polling strategy.

In addition to the simple pipeline microbenchmark discussed in the previous section, we tested a set of streaming

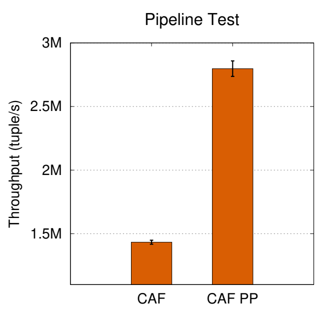

Fig. 9. Maximum throughput of the *Pipeline* microbenchmark.

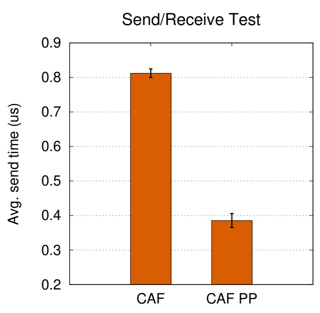

Fig. 10. Average send time of Prod-Cons. microbenchmark.

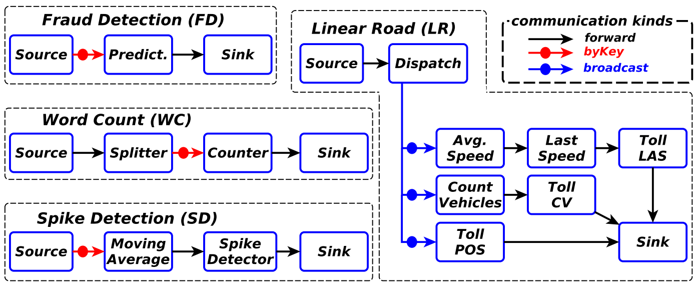

Fig. 11. Applications used in the evaluation.

applications typically used to evaluate Stream Processing Engines, namely *Word Count*, *Fraud Detection*, *Spike Detection*, *Linear Road*, briefly described in the following.

Our first test was to evaluate the improvement of using *SeqNode*s instead of *SeqActor*s in the three-stage pipeline microbenchmark. Fig. 9 compares the maximum throughput (tuples/s) obtained by the the two versions, CAF implementing *Pipeline* (*SeqActor*,*SeqActor*,*SeqActor*); CAF PP implementing *Pipeline* (*SeqNode*,*SeqNode*,*SeqNode*), of the same operators. A large part of the overhead discussed in Sec. 3 has been removed. The difference with the baseline (called *Thread* in Fig. 4), is now about 15% (2.8 M *vs* 3.3 M tuples/s), which means that there are still a small margin for fine-tuning our *SeqNode* implementation. Concerning the cost for exchanging a message between a Producer and a Consumer Actor, Fig. 10 shows that the *SeqNode*-based implementation (i.e., CAF PP) reduces the average time from 0.81 µs to 0.38 µs.

Applications. We considered four streaming applications whose operators graphs are sketched in Fig. 11[5]. The figure also shows the kind of communication between operators: *forward*, *byKey* and *broadcast*. These communication attributes are meaningful if the next operator is replicated using a *Farm* pattern. The *forward* communication is the default one. It states that an input tuple can be assigned to any replicas. We implemented this communication mode with the *round-robin* policy strategy of the *Farm* pattern. The *byKey* distribution allows sending all input tuples with the same key attribute (i.e., a specific field of the tuple) to the same operator replica. Finally, the *broadcast* distribution duplicates the tuple and sends it to all next operators.

Fraud Detection (FD) applies a Markov model to compute the probability of a credit card transaction to be a fraud. *Spike Detection* (SD) finds the spikes in a stream of sensor readings using a moving-average operator and a filter. *Word Count* (WC) counts the number of instances of each word in a text file. An operator splits the lines into words; a second operator counts the word instances. *Linear Road* (LR) emulates a tolling system for the vehicle expressways. The system uses a variable tolling technique accounting for traffic congestion and accident proximity to calculate toll charges.

In Fig. 12, 13, 14, 15 we reported the throughput obtained by implementing the applications' operators (one replica per operator) by using the two implementation skeletons for the *Sequential* pattern (*SeqActor vs SeqNode*, labeled with CAF and CAF PP, respectively). All tests have been executed 20 times for 60 seconds. The average value obtained and the error-bar are reported in the figures. The throughput has been measured in the *Sink* operator varying the input data rate in the *Source*. As long as the data rate is relatively low, there are no significant differences between the two implementation skeletons, even though the end-to-end latency is higher for the *SeqActor*-based implementation (not reported here for space reasons). Conversely, with high data rates, the difference between the two versions is increasing significantly. *Word Count* presents the biggest difference. This is due to the very high data rate produced by the `Splitter` operator that gets in input a line from the `Source`, and produces in output all words it contains, thus multiplying the nominal input data rate.

Fig. 16 shows the performance improvement obtained for all applications by the *SeqNode*-based implementations. In this case, for both implementations, we ran the *Source* operator at maximum speed. In addition, they use the `send_next_if` primitive. The "queue length value" is set to 1K elements, and if reached, the *Source* waits for some nanoseconds before retrying the send. In this way, the internal pipeline operators are not overloaded and the entire system stabilizes after few seconds of execution. In this case we executed all applications for 10 minutes. As expected, the improvement of the CAF PP version using *SeqNode*s is significant, more than 2× in all application but *Fraud Detection* where it is about 30%.

Finally, the table in Fig. 17 reports the results obtained in all applications by replicating all streaming operators. Instead of finding the best replication degree for each operator (operators have different execution times, and some of them may need more replicas while others do not need to be replicated at all), we decided to equally replicate all of them until we fill up all logical cores of the server considered (48 in our case). The replication degree is specified in the table. This

[5]The C++ source code is publicly available in GitHub: https://github.com/ParaGroup/StreamBenchmarks.

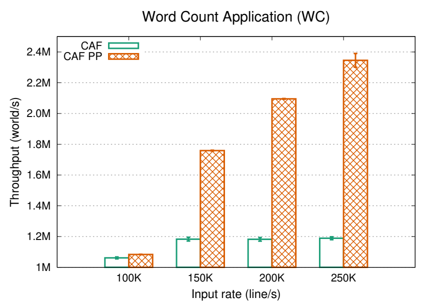

Fig. 12. Throughput varying the input rate for the *Word Count* application.

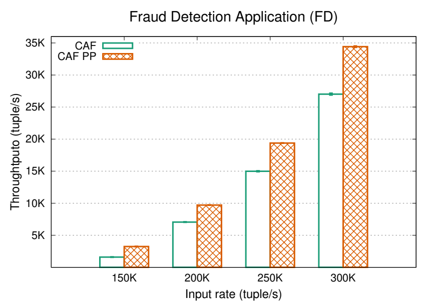

Fig. 13. Throughput varying the input rate for the *Fraud Detection* application

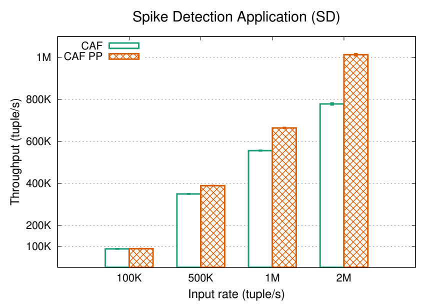

Fig. 14. Throughput varying the input rate for the *Spike Detection* application

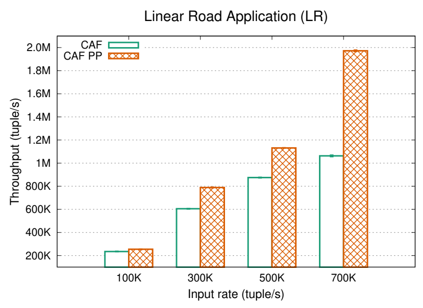

Fig. 15. Throughput varying the input rate for the *Linear Road* application

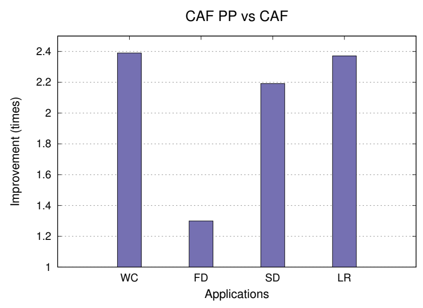

Fig. 16. Performance improvement of CAF PP vs CAF for WC, FD, SP, LR applications. The tests were executed at maximum rate in the *Source.*

	WC	**FD**	**SD**	**LR**
Operators	4	3	4	9
Replicas	×12	×16	×12	×5
CAF	2.6M	327K	1.6M	1.0M
CAF PP	8.1M	560K	9.7M	2.1M
Improvement	3.1	1.7	6	2

Fig. 17. Applications throughput (tuples/s) obtained replicating all operators. The improvement is the ratio between CAF PP and CAF.

way, we can evaluate the behavior of the two implementation skeletons under very high data rates, since all *Source* replicas execute at maximum speed. In this test we used for both versions the implementation skeleton for the *Farm* pattern that removes the routing Actor between two consecutive *Farm* PPs. For all applications, we obtained a performance improvement in terms of throughput, and as expected, the relative distance between the two implementation skeletons (i.e., *SeqActor vs SeqNode*) further increased in all applications but *Linear Road*, where the relative distance remains about the same due to its peculiarities.

All in all, at high data rates, the new implementation skeletons of our PPs provide a definite performance boost without compromising the AM's message-passing semantics.

6 Related Work

The Actor Model provides important guarantees to deadlock-freedom and data-race-freedom. However, on shared-memory systems, those guarantees come at the price of some extra performance overheads. Combining the Actor Model with shared-memory for performance is efficient but may introduce data-races. Several research works focused on improving the performance of the Actor Model without losing its important guarantees. Some tried to provide more efficient run-time mechanisms to speed up the execution of Actor-based applications [14, 31, 45, 46], others, instead provide high-level abstractions to support the distinctive features of the Actor Model [19, 23, 34, 40, 44].

Some works focused on improving the AM messaging system. ACTOR4J [15] is a new AM implemented in JAVA that focuses on optimizing message exchange among Actors. ACTOR4J differently from AKKA [10] moves the Actor queue to the underline native executor, thus the Actor is bound to that executor along with other concurrent Actors. Using this approach the implementation can optimize message queue at the level of the underline executor guaranteeing low latency.

SALSA LITE [28] is a run-time the AM language SALSA. It has been designed to efficiently implements the basic features of the Actor Model, i.e. message-passing and dynamic Actors creation. SALSA LITE is highly optimized for multi-cores and proposes a non-transparent execution of Actors where the user should manually assign Actors to executors.

Selectors [35] in an extension of the AM. The *Selectors* manages multiple incoming queues that can be activated or deactivated. A deactivated queue may continue to receive messages, but the Selectors will temporarily ignore it. The authors claimed that the use of the SELECTORS concept makes synchronization and coordination patterns more natural to implement (e.g., synchronous request-reply, producer-consumer with bounded buffer).

The CAL Actor Language [29] attempted to integrate the data-flow programming model with Actors. CAL has been adopted in the standardization effort of the MPEG Reconfigurable Video Coding Framework [17].

Akka [10] is one of the most used Actor frameworks. In Akka, the programmer can choose the type of queue to use for the mailbox of each Actor [5]. There is a set of default queues with different features, e.g. bounded or unbounded queue, priority-based queues, and it is even possible to supply a custom queue. Moreover, Akka provides mechanisms to replicate Actors by using *Routers* [4]. A *Router* provides a set of distribution policies, one of them being the possibility to send the message to the Actors with the smallest queue length. This feature is supported regardless of the kind of mailbox used by the individual Actor. Every Akka mailbox implementation provides an API to get the queue length. Moreover, Akka has an extension, not based on the AM, designed for streaming called Akka Stream. The system is based on the *Reactive Stream* project and implements a static graph of operators with typed messages.

Instead, the CAF Stream experimental extension of CAF, has a similar design of Akka Stream, but (for now), it does not provide any high-level structure for building networks of operators. It maintains a close connection with CAF Actors.

Recently, there has been an interest in developing SPEs for scale-up servers [39, 48]. However, most of them are based on the JVM or use the batching model. The WindFlow library [38], written in modern C++, implements the continuous streaming model and leverage PP abstractions for complex streaming operators. Its run-time is based on the thread-based FastFlow library [9].

To the best of our knowledge, there were no previous attempts to integrate optimized stream-oriented PPs within the Actor Model.

7 Summary

In this work, we studied the use of the Actor Model in high data rate streaming computations on multi-cores. We used C++ Actor Framework as a reference implementation of the Actor Model. We discussed the performance limits of Actor-based implementation in high-throughput streaming applications. Then, we propose a new *implementation skeleton* of both the *Sequential* and *Farm* Parallel Patterns for CAF, which have been specifically optimized for high-throughput and low-latency data streaming computations implemented with Actors. The results obtained by testing a small set of well-known benchmarks demonstrate the benefits of using the new implementations, which can bring a performance boost of more than 2× on state-of-the-art scale-up servers. The optimizations introduced are transparent to the CAF Actor programmer that uses PPs.

References

[1] 2020. Apache Flink. https://flink.apache.org/

[2] 2020. Apache Storm. http://storm.apache.org/

[3] 2020. CAF Streaming (experimental). https://actor-framework.readthedocs.io/en/latest/Streaming.html

[4] 2020. Classic Routing • Akka Documentation. https://doc.akka.io/docs/akka/2.6.8/routing.html

[5] 2020. Mailboxes • Akka Documentation. https://doc.akka.io/docs/akka/2.6.8/typed/mailboxes.html

[6] Gul Agha. 1984. *Actors: A Model of Concurrent Computation in Distributed Systems*. Ph.D. Dissertation. https://www.researchgate.net/publication/37597732_ACTORS_A_Model_of_Concurrent_Computation_in_Distributed_Systems

[7] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. 1997. A foundation for actor computation. *Journal of Functional Programming* 7, 1 (Jan. 1997), 1–72. https://doi.org/10.1017/S095679689700261X

[8] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. 2012. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing (LNCS, Vol. 7484)*. Springer, 662–673. https://doi.org/10.1007/978-3-642-32820-6_65

[9] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. Fastflow: High-Level and Efficient Streaming on Multicore. In *Programming multi-core and many-core computing systems*. John Wiley & Sons, Ltd, 261–280. https://doi.org/10.1002/9781119332015.ch13 Section: 13 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119332015.ch13.

[10] Jamie Allen. 2013. *Effective Akka: Patterns and Best Practices*. " O'Reilly Media, Inc.".

[11] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. 2013. Scale-up vs Scale-out for Hadoop: Time to Rethink?. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. Association for Computing Machinery, New York, NY, USA, Article 20, 13 pages. https://doi.org/10.1145/2523616.2523629

[12] Joe Armstrong and "Ericsson Telecom Ab". 1997. The development of Erlang. *Association for Computing Machinery* 32, 8 (1997), 196–203. https://doi.org/10.1145/258948.258967

[13] Keyvan Azadbakht, Frank S. de Boer, Nikolaos Bezirgiannis, and Erik de Vink. 2020. A formal actor-based model for streaming the future. *Science of Computer Programming* 186 (2020), 102341. https://doi.org/10.1016/j.scico.2019.102341

[14] Saman Barghi and Martin Karsten. 2018. Work-Stealing, Locality-Aware Actor Scheduling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Vancouver, BC, 484–494. https://doi.org/10.1109/IPDPS.2018.00058

[15] David Alessandro Bauer and Juho Mäkiö. 2018. Actor4j: A Software Framework for the Actor Model Focusing on the Optimization of Message Passing. (2018), 10.

[16] Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2016. Orleans: Distributed Virtual Actors for Programmability and Scalability. (2016), 13.

[17] Shuvra S. Bhattacharyya, Johan Eker, Jörn W. Janneck, Christophe Lucarz, Marco Mattavelli, and Mickaël Raulet. 2011. Overview of the MPEG Reconfigurable Video Coding Framework. *Journal of Signal Processing Systems* 63, 2 (May 2011), 251–263. https://doi.org/10.1007/s11265-009-0399-3

[18] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *17th Inter. Conf. on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. ACM, 72–81. https://doi.org/10.1145/1454115.1454128

[19] István Bozó, Kevin Hammond, Viktoria Fordós, Zoltán Horvath, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, and Christopher Brown. 2014. Discovering parallel pattern candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang - Erlang '14*. ACM Press, Gothenburg, Sweden,

13–23. https://doi.org/10.1145/2633448.2633453

[20] Colin Campbell and Ade Miller. 2011. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures* (1st ed.). Microsoft Press, USA.

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

[22] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. 2013. Native actors: a scalable software platform for distributed, heterogeneous environments. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control - AGERE! '13*. ACM Press, Indianapolis, Indiana, USA, 87–96. https://doi.org/10.1145/2541329.2541336

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

[24] Murray Cole. 2004. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.* 30, 3 (2004), 389 – 406. https://doi.org/10.1016/j.parco.2003.12.002

[25] Adam L. Davis. 2019. *Akka Streams*. Apress, Berkeley, CA, 57–70. https://doi.org/10.1007/978-1-4842-4176-9_6

[26] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2016*. ACM Press, Amsterdam, Netherlands, 31–40. https://doi.org/10.1145/3001886.3001890

[27] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. 2017. Bringing Parallel Patterns Out of the Corner: The P3ARSEC Benchmark Suite. *ACM Trans. Archit. Code Optim.* 14, 4, Article 33 (Oct. 2017), 26 pages. https://doi.org/10.1145/3132710

[28] Travis Desell and Carlos A. Varela. 2014. SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency. In *Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*. Springer Berlin Heidelberg, Berlin, Heidelberg, 144–166. https://doi.org/10.1007/978-3-662-44471-9_7

[29] Johan Eker and J Janneck. 2003. *CAL language report: Specification of the CAL actor language*. December.

[30] August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming* 46, 1 (Feb. 2018), 62–80. https://doi.org/10.1007/s10766-017-0490-5

[31] Emilio Francesquini, Alfredo Goldman, and Jean-François Méhaut. 2013. Improving the Performance of Actor Model Runtime Environments on Multicore and Manycore Platforms. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2013)*. Association for Computing Machinery, New York, NY, USA, 109–114. https://doi.org/10.1145/2541329.2541342 event-place: Indianapolis, Indiana, USA.

[32] Buğra Gedik. 2014. Generic Windowing Support for Extensible Stream Processing Systems. *Softw. Pract. Exper.* 44, 9 (Sept. 2014), 1105–1128. https://doi.org/10.1002/spe.2194

[33] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, Vol. 3. Stanford Research Institute, 235.

[34] Shams M. Imam and Vivek Sarkar. 2012. Integrating Task Parallelism with Actors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 753–772. https://doi.org/10.1145/2384616.2384671 event-place: Tucson, Arizona, USA.

[35] Shams M. Imam and Vivek Sarkar. 2014. Selectors: Actors with Multiple Guarded Mailboxes. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control - AGERE! '14*. ACM Press, Portland, Oregon, USA, 1–14. https://doi.org/10.1145/2687357.2687360

[36] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[37] Timothy G Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for parallel programming*. Pearson Education.

[38] G. Mencagli, M. Torquati, D. Griebler, M. Danelutto, and L. G. L. Fernandes. 2019. Raising the Parallel Abstraction Level for Streaming Analytics Applications. *IEEE Access* 7 (2019), 131944–131961.

[39] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2017. StreamBox: Modern Stream Processing on a Multicore Machine. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 617–629.

[40] Aleksandar Prokopec and Martin Odersky. 2015. Isolates, channels, and event streams for composable distributed programming. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) - Onward! 2015*. ACM Press, Pittsburgh, PA, USA, 171–182. https://doi.org/10.1145/2814228.2814245

[41] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.".

[42] Luca Rinaldi, Massimo Torquati, Daniele De Sensi, Gabriele Mencagli, and Marco Danelutto. 2020. Improving the Performance of Actors on Multi-cores with Parallel Patterns. *International Journal of Parallel Programming* (June 2020). https://doi.org/10.1007/s10766-020-00663-1

[43] Luca Rinaldi, Massimo Torquati, Gabriele Mencagli, Marco Danelutto, and Tullio Menga. 2019. Accelerating Actor-Based Applications with Parallel Patterns. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, Pavia, Italy, 140–147. https://doi.org/10.1109/EMPDP.2019.8671602

[44] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 2018. Chocola: integrating futures, actors, and transactions. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2018*. ACM Press, Boston, MA, USA, 33–43. https://doi.org/10.1145/3281366.3281373

[45] Massimo Torquati, Tullio Menga, Tiziano De Matteis, Daniele De Sensi, and Gabriele Mencagli. 2018. Reducing Message Latency and CPU Utilization in the CAF Actor Framework. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, Cambridge, 145–153. https://doi.org/10.1109/PDP2018.2018.00028

[46] Phil Trinder, Olivier Boudeville, and Francesco et al. Cesarini. 2017. Scaling Reliably: Improving the Scalability of the Erlang Distributed Actor Platform. *ACM Transactions on Programming Languages and Systems* 39, 4 (Aug. 2017), 1–46. https://doi.org/10.1145/3107937

[47] Vaughn Vernon. 2016. *Reactive messaging patterns with the Actor model: applications and integration in Scala and Akka*. Addison-Wesley, New York.

[48] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. 2017. Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 659–670. https://doi.org/10.1109/ICDE.2017.119