Accelerating Actor-based Applications with Parallel Patterns

Luca Rinaldi*, Massimo Torquati*, Gabriele Mencagli*, Marco Danelutto*, Tullio Menga†

*Department of Computer Science, University of Pisa, Pisa Italy

{ luca.rinaldi, torquati, mencagli, danelutto }@di.unipi.it

†ATS Advanced Technology Solutions S.p.A., Milan, Italy

tullio.menga@atscom.it

***Abstract*— Parallel programmers mandate high-level parallel programming tools allowing to reduce the effort of the efficient parallelization of their applications. Parallel programming leveraging parallel patterns has recently received renovated attention thanks to their clear functional and parallel semantics. In this work, we propose a synergy between the well-known Actors-based programming model and the pattern-based parallelization methodology. We present our preliminary results in that direction, discussing and assessing the implementation of the *Map* parallel pattern by using an Actor-based software accelerator abstraction that seamlessly integrates within the C++ Actor Framework (CAF). The results obtained on the Intel Xeon Phi KNL platform demonstrate good performance figures achieved with negligible programming efforts.**

I. INTRODUCTION

The proliferation of multi/many-core systems, especially in the consumer hardware, has produced a game-changing for the computing industry, which today is relying on parallel processing to improve applications performances [1]. Parallel programs are far more challenging to design, write, debug, and tune than sequential ones, and it is common for parallel programmers to face deadlock, starvation and data-race issues.

Over the years, multiple programming models have been proposed to cope with the parallel programming issues. One of these is the Actor model formulated by Hewitt, Bishop, and Steige in 1973 [2]. Recently, it has gained renewed attention as a powerful approach to the problems of concurrency not only from the theoretical viewpoint. This is mainly due to the development of Cloud Computing and other dynamic and heterogeneous distributed environments [3] and also to the success of programming languages such as Erlang [4] and Scala [5], two implementations inspired to the Actor model.

The Actor model formulation describes entities called *Actors* that execute independently and concurrently. Actors do not explicitly share states and communicate only via asynchronous message-passing through unbounded communication channels. Since Actors are self-contained concurrent entities, they avoid race conditions by design. Moreover, Actors do not explicitly support synchronization mechanisms relying only on asynchronous messages, thus preventing deadlock situations. The lack of shared states makes the Actor model particularly attractive for distributed and heterogeneous platforms.

Preprint version of the PDP 2019 conference paper.

Nevertheless, the Actor model, like other similar models (e.g., the task-based model) follows a non-structured approach to parallel programming, which leaves the users free to build their concurrent system, without any performance guarantee. This aspect, from the one hand, allows great flexibility in the software development process, on the other hand, it does not offer to the programmer any methodology for solving performance related issues. It is common to encounter, in Actor-based applications, Actors that, under particular conditions, become bottlenecks thus reducing the overall system performance. These situations are not easy to discover and to eliminate. Usually, the programmer has to manually try to split the computation over multiple Actors to mitigate the bottleneck issue.

A different approach for dealing with issues related to parallelism exploitations is to use *parallel patterns* [6]. Parallel patterns such as *Map*, *Ruduce* and *Pipeline* add semantics information about the way parallelism has to be exploited, offering a clear functional and parallel semantics. The parallelization methodology based on parallel patterns arises from the attempts of capturing the best practices through which parallel software can be organized, designed, and implemented. Parallel patterns are becoming an increasingly popular programming approach for writing high-level and efficient parallel applications targeting multiple platforms, from multi-cores [7], to distributed systems [8], to hardware accelerators such as GPUs [9] and FPGAs [10].

Our research aim is to study how to combine the Actor-based programming model with the structured parallel programming methodology based on parallel patterns to take the best of the two worlds. In this paper, we propose a first attempt of integrating the two approaches by using the CAF framework [11], [12] as an implementation of the Actor model in modern C++.

The contribution of this work is twofold:

- we propose an Actor-based software accelerator that implements a *Map* parallel pattern that may be instanced and used by multiple CAF Actors to speed up map-like computations.
- we introduce the *thread-to-core affinity* feature in the CAF run-time system to enable the possibility to separate the threads used to implement the accelerator from the threads used in the CAF run-time to execute the Actors.

The rest of the paper is organized as follows. Section II

provides the background materials related to the Actor model, the parallel pattern methodology and the CAF framework. Section III describes the design and implementation of the *Map* accelerator proposed in this work, and the *thread-to-core affinity* implementation in the CAF run-time system. Section IV shows the preliminary performance evaluation of the Map accelerator. Finally, Section V contains the related works and Section VI outlines the conclusions and possible future work.

II. BACKGROUND

A. The Actor model

The Actor model was proposed by Hewitt, Bishop and Steiger in their seminal work on 1973 [2]. It was initially developed to be used for artificial intelligence research purposes to modelling system with thousands of independent processors each having a local memory and connected through some high-performance network. More recently the Actor model gained new interest and started to be adopted in the contest of multi/many-core architectures [13].

The Actor model is centered around the *Actor* concept. An Actor is a concurrent entity that:

- interacts with other Actors through messages.
- may create a finite number of new Actors,
- may dynamically change its internal behavior.

Every Actor has an input channel (called *mailbox*) where it receives messages from other Actors. For each message, it performs a local computation based on its private internal state and on the message just received; it may generate new messages toward on or more Actors (included itself).

The interactions among Actors are based on asynchronous, unordered, fully distributed, address-based messaging. To achieve full asynchronicity, and thus preventing deadlock, each Actor has an input mailbox of unbounded capacity. However, no guarantee is given about the ordering in which the messages are processed by an Actor. There is no upper-bound guarantee on the time needed by the system to reach a stable state, i.e. a state where all the messages have been processed and no other messages have to be delivered. The Actor model only guarantees that all messages will be eventually processed. One fundamental aspect of the Actor model is that there is no global state and no central entity managing the whole system. The computation is a partial ordering of a sequence of transitions from one local state to another. Unordered events may be executed in parallel, therefore the run-time system can compute the new local state of multiple Actors concurrently. The flow of events in an Actors system forms an activation sub-ordered tree expressing, for every event, a causality order with a finite path back to the initial event. Different branches of the activation sub-ordered tree represent chains of parallel events. The Actor model enforces strict locality by allowing Actors to build their knowledge about the rest of the system only through messages, including addresses of other Actors. Indeed, every Actor maintains a dynamic list of acquaintance Actors, representing its partial view of the system. An Actor enlarges the acquaintance list if it spawns a new Actor, if it receives a message from an unknown Actor, or if it receives the address of other Actors via incoming messages.

B. C++ Actor Framework (CAF)

The C++ Actor Framework (CAF) [11], [12] allows the development of concurrent programs based on the Actor model leveraging modern C++ features. Differently from other well-known implementations of the Actor model, such as Erlang [4] and Akka [14], which use virtual machine abstractions, CAF is entirely implemented in modern C++ thus compiling directly into native machine code.

CAF applications are built decomposing the computation in small independent work items that are spawned as Actors and executed cooperatively by the CAF run-time. Actors are modeled as lightweight state machines that are mapped onto a pre-dimensioned set of run-time threads called Workers. Instead of assigning dedicated threads to Actors, the CAF runtime includes a scheduler that dynamically allocates ready Actors to Workers. Whenever a waiting actor receives a message, it changes its internal state and the scheduler assigns the Actor to one of the Worker threads for its execution. As a result, the creation and destruction of actors is a lightweight operation.

Actors that use blocking system calls (e.g., I/O functions) can suspend run-time threads creating either imbalance in the workload of system threads or their starvation. The CAF programmer can explicitly *detach* Actors so that the Actor will be associated with a dedicated system thread. A particular kind of detached actor is the *blocking actor*. *Detached* and *blocking* Actors are not as lightweight as default event-based CAF Actors.

In CAF, Actors are created using the *spawn* function, which creates an Actor from functions, C++ lambdas, and class instances. It returns a network-transparent Actor handle corresponding to the Actor address. Communication happens via explicit message-passing by using the *send* command. Messages are buffered into the mailbox of the receiver Actor in arrival order. The response to an input message can be implemented by defining *behaviors*. The handler function signature is used to identify different behaviors.

C. Parallel patterns

Parallel patterns [6] are schemas of parallel computations that recur in the realization of many algorithms and applications for which parametric implementations are available Such well-known parallel structures have a rigorous semantics with an associated cost model that allows evaluating their profitability. Some notable examples of parallel patters are *Map*, *reduce*, *pipeline*, *task-farm*, *divide-and-conquer*, *stencil*, and *parallel-for*. The programming approach based on parallel patterns is called *structured parallel programming* [15]. This approach provides the parallel application programmer with a set of predefined, ready-to-use parallel abstractions that may be directly instantiated, alone or in composition with, to model the complete parallel behavior of the application. This raises the level of abstrac-

tion by ensuring that the application programmer does not need to deal with parallelism exploitation issues and low-level architectural details during application development. Instead, these issues are efficiently managed using state-of-art techniques by the system programmer while designing the development framework and its associated run-time. The use of parallel patterns in the development of applications provides several advantages both concerning time-to-solution and the automatic or semiautomatic applicability of different optimization strategies. This last aspect is usually manually enforced in programming models that do not use a pattern-based approach, such as the Actor model.

Combinations of parallel design patterns and their concrete implementations called also *algorithmic skeletons* are used in different parallel programming frameworks such as SkePU [9], Muesli [16], FastFlow [17], Delite [18] and GrPPI [19], just to mention few of them. Other frameworks such as Google MapReduce [8] are instead built around a single powerful parallel pattern.

III. PATTERN-BASED SOFTWARE ACCELERATOR

A. Design

The Actor model promotes an unstructured approach to parallel programming. A high number of concurrent activities embedded into Actors cooperate by exchanging messages to solve a given problem. Actors are dynamic entities, whose behavior may evolve over time and new connections between Actors may be dynamically activated. The high degree of freedom and flexibility offered by the Actor model may easily lead to building complex Actor-based topologies which are difficult to modify, debug and optimize.

To deal with these issues, we envision a synergy between the Actor model and the structured parallel programming approach based on parallel patterns. We propose to design and implement some essential parallel patterns as fundamental components that can be used as "Actors' accelerator". Parallel patterns (and possibly their compositions) offer a clear functional and parallel semantics that simplifies parallel programming. Moreover, it relieves the application programmers from the responsibility of designing and implementing efficient and well-known parallel components, allowing them to concentrate on the business logic of the application considered.

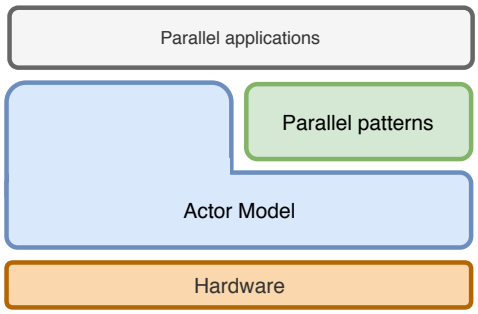

Fig. 1: Layered design of the synergy between the Actor model and the parallel patterns approach.

As shown in Fig. 1, our proposal leverages a software layer providing a set of parallel patterns implemented by using Actors that seamlessly coexist with the Actors used to implement applications. The user may use parallel patterns together with Actors to take advantage of both the flexibility of the Actor model and the performance offered by the specific parallel pattern implementation.

In the following, we will show the design and the implementation of a software accelerator for *Map* pattern.

The *Map* pattern is a data-parallel paradigm that applies the same function to every element of an input collection. The *Map* is defined as $(\texttt{map}\ f) : \alpha\ \texttt{collection} \rightarrow \beta\ \texttt{collection}$ and computes a function $f : \alpha \rightarrow \beta$ over all items of an input collection whose elements have type α. The output produced is a collection of items of type β where $y_i : \beta, y_i = f(x_i)$ for each itemi x_i of the input collection. The precondition is that all items of the input collection are independent and can be computed in parallel.

If we consider the case of an Actor that has to execute a *Map* computation on a large input collection, its service time is given by the time needed to compute the single element of the collection multiplied by the number of elements. If this Actor offloads the computation it has to execute to a parallel implementation of the *Map* pattern, its service time could be reduced *ideally* to the time needed to compute a single element.

We designed the *Map* accelerator as a set of Actors with a predefined communications topology. It can be instantiated by any Actor and its address can be shared with other Actors via messages. Once spawned, the accelerator waits for incoming requests. Multiple Actors may send a request containing both the input data collection and the function to compute in parallel. The accelerator will send back the result as soon as it is available. The *Map* accelerator can also be used in streaming computations where the generic Actor needs to speed up its local data-parallel computation on the incoming input data.

An important point to consider is the potential conflicts that the threads implementing the software accelerators may have with the run-time threads used for executing the application Actors. To minimize the interference introduced by the software accelerator, we modified the run-time of the CAF framework to control the placement of system threads on different cores of the target platform. We call this new feature *thread-to-core affinity*. By using this new low-level feature, we can confine the run-time threads used to implement the Actors of the *Map* accelerator to a restricted set of machine cores.

B. Implementation

The implementation of the *Map* accelerator has been done at two different level. At the bottom level, we modified the CAF run-time to introduce the possibility to manage the *thread-to-core affinity* for the run-time system threads. This allows to control the mapping of different CAF threads used to execute Actors, for example to confine the threads used for implementing the *Map* accelerator on a subset of the machine

cores. At the top level, we designed and implemented the software accelerator and its API by using the Actor model implemented by CAF.

Affinity control implementation. With the terms *thread-to-core affinity* (or simply *thread affinity*) we refer to the possibility to control on which logical core(s) a given thread can be executed by the OS. This prevents the OS to move the thread on a different set of cores thus reducing potential noise introduced by the OS scheduler.

The CAF framework defines different types of threads: the ones used for implementing the *thread-pool* in charge of executing the *event-based Actors*, the ones used to execute *detached Actors* and those used for executing the *blocking Actors*. We defined a new system configuration parameter that allows to statically specify on which cores the different kinds of CAF run-time threads have to be executed by the OS. The set of cores can be selected by using an *affinity string* (*affinitystr*), i.e. a string whose format respect the following grammar (*coreid* is a valid core identifier):

$$
\begin{aligned}
range &::= coreid \mid coreid - coreid \\
rangelist &::= range \mid range, rangelist \\
group &::= < rangelist > \\
grouplist &::= group \mid group\ grouplist \\
affinitystr &::= grouplist
\end{aligned}
$$

The *affinitystr* is composed by a set of groups enclose in angle brackets (`< >`). A group hosts a collection of cores separated by commas (`,`) or a range of them delimited with a single dash (−). The CAF run-time has been modified in such a way to read the *affinity string* and to execute the proper system calls for setting the thread affinity. For example, the *affinitystr* `"<0> <2-4> <1,5>"` allows placing the first thread spawned by CAF on the core with id 0, the second thread on cores 2, 3 and 4, and the third thread on cores 1 and 5. The next thread spawned will be placed again on the first group, i.e. core 0.

This new feature permits to separates CPU resources among different kinds of CAF threads, and particularly allows avoiding overlap between the CAF run-time thread used for event-based Actors from other threads.

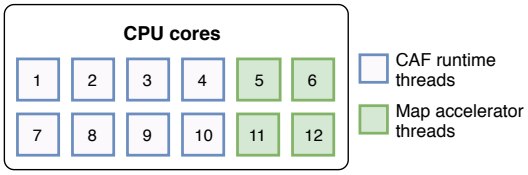

Fig. 2: Example of *thread-to-core affinity* in CAF.

Figure 2 shows a simple example where CPU cores have been partitioned between the CAF run-time threads and the threads used for running the Actors implementing the *Map* accelerator.

Map implementation. We implemented the Map accelerator by using two different Actors: the *scatter/gather* and the *Worker*, the latter replicated a number of times. They are connected according to a predefined communication schema (see the right-hand side of Fig. 3). The *scatter/gather* Actor manages both incoming requests coming from "external" Actors and the partial results coming from the pool of "internal" *Worker* Actors. The n Workers of the pool, apply the Map function on disjoint portions of the input data collection producing n partial results that are then assembled by the manager *scatter/gather* Actor.

The generic Worker Actor, wait for an incoming chunk of data elements and the function to apply to each item of the collection. Thanks to the *zero-copy* feature of the CAF run-time, all Workers read from the same data collection and then each of them creates an internal copy of the collection storing only the computed results. The manager Actor receives the computed chunks and creates a new data collection with the results that will be eventually sent back to the sender.

Figure 3 shows a code snippet in which the *Map* accelerator is instanced and used by a single Actor. The logical Actor schema produced by the code snipped is sketched in the right-hand side of the figure. The Actor creates a new *Map* accelerator instance and starts to offload data read from a file. Then the Actor asynchronously sends the result obtained from the accelerator to another Actor (*next actor* in the figure).

In particular, in the line 2 a new instance of the *Map* accelerator is created by using the CAF `spawn` function passing the number of Workers to be used. Line 7 sends the request to the accelerator by using the `request` CAF function. A vector of integers and the lambda function defined at line 4 is provided as input arguments. The sender Actor creates an asynchronous handler for the promise of the result. When the *Map* accelerator completes the execution, it sends back to the Actor the result that is then used in the callback function defined at line 9.

It is worth noting that the *Map* accelerator seamlessly integrates with the Actor model implementation provided by the CAF framework. CAF's Actors can spawn and interact with the accelerator in the same way they communicate with each other.

To use the *thread-to-core affinity* feature with the aim of separating the resources used for the accelerator from the ones used for the CAF run-time, the *Map* accelerator spawns its internal Workers as detached Actors. The scatter/gather Actor is instead executed as event-based Actor since its associated computational cost is low.

IV. EXPERIMENTAL RESULTS

In this Section, we test the CAF implementation of the *Map* accelerator discussing both a simple synthetic *Map* benchmark and also a modified version of the CAF Latency Benchmark described in [20].

All tests were executed on a Intel Xeon Phi 7210 many-core platform (codename Knights Landing, KNL). The KNL is equipped with 32 tiles (each with two cores) working at 1.3GHz, interconnected by an on-chip mesh network. Each core (4-way SMT) has 32 KB L1d private cache and a L2 cache of 1 MB shared with the sibling core on the same tile.

```
/* Spawn a map accelerator instance */
auto map_instance = caf::spawn(map,  5);
/* Declare the Map function */
auto F = [](int el){ return el + 1;};
for (auto vec : read_from_disk()){
  /* Offload the computation to the accelerator */
  caf::request(map_instance, F, vec).then(
    /* Async receive */
    [=] (std::vector<int> result) {
      send(next_actor, result);  //send the result
  });
}
```

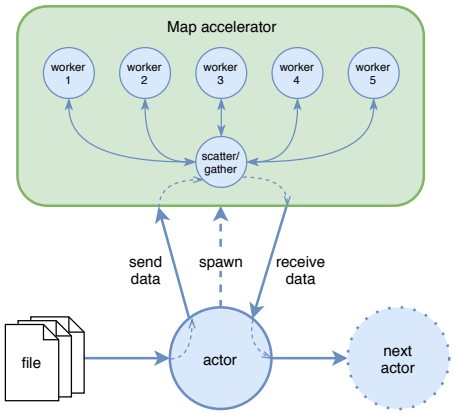

Fig. 3: An usage example of the *Map* accelerator (left-hand side). Logical schema of the example (right-hand side).

The machine is configured with 96 GB of DDR4 RAM with 16 GB of high-speed on-package MCDRAM configured in cache mode. The machine runs CentOS 7.2 with Linux kernel 3.10.0 and the GCC compiler version 7.3.0. The code was compiled with the `-O3` optimization flag. All plots report the average value obtained by five distinct runs.

The first benchmark considers a data-parallel computation on a matrix A of size $N \times M$. The program spawns an Actor that computes, for each row of the input matrix, a function f on each element of the row and then it sums up all elements of the row. The symbolic computation of the i-th row is the following: $\sum_{j=0}^{M} f(A[i,j])) \ \forall i \in [0, N)$.

We parallelized the computing Actor by spawning the *Map* accelerator and offloading to the accelerator the computation of the function f over the matrix rows. The computing Actor then executes the reduce part locally.

The benchmark has been executed with an input matrix of size 100×5000, and the function f executes a synthetic computation of about $200\mu s$ on each element of the input row. The test has been run with and without the thread-to-core affinity configuration to evaluate the performance improvement of isolating the accelerator threads from the CAF run-time threads.

Figure 4 shows the execution time (in seconds) and the scalability of the test. The total number of threads used is fixed to 64 and they are mapped, by using the `taskset` Linux command, to the 64 physical cores of the machine. For the test, the system threads have been split into two subsets: 1) a set of threads assigned to the *Map* accelerator, and 2) a second set assigned to the CAF run-time threads. In the plot, the number of *Map*'s Workers used is reported in the bottom x-axis, while the number of threads assigned to the CAF run-time system is reported in the top x-axis. As can be seen, the use of the *Map* accelerator allows obtaining a scalability of about 50 with 63 *Map* Workers. Moreover, the version that isolates the accelerator threads from the other run-time threads captures a non-negligible performance advantage for the execution time.

As a second test, we considered the CAF Latency Benchmark [20]. It aims at measuring the message latency of CAF's Actors considering either single pipeline of Actors or multiple replicas of pipeline chains each one having a fixed number of Actors (see Fig. 5). A *Rate generator* Actor generates messages at a given constant rate. The *Result collector* Actor, collects all messages and compute the average message latency.

To evaluate the *Map* accelerator implementation, we modified the CAF Latency Benchmark by adding a new type of Actor in the pipeline that instead of just forwarding the message to the next Actor, it executes a data-parallel computation on the input message (the schema of the modified benchmark is shown in Fig. 6). There is only one computing Actor for each pipeline chain. These "heavy-weight" Actors are the bottlenecks of the pipelines potentially producing a significant increase in the average message latency. The objective of this benchmark is to show how the message latency can be reduced by parallelizing the compute intensive Actors by using the *Map* accelerator. To this end, a single instance of the *Map* accelerator is created at the program start-up, and all computing Actors share its reference. In this way, they can offload their computation to the parallel accelerator to decrease their service time.

We tested the case of 100 pipeline chains each one with 8 Actors. The message rate is fixed to $1,000$ messages per second. The computing Actors work on an input collection of $5,000$ elements. The average execution time per item is about $1\mu s$. The benchmark lasts 15 seconds.

Figure 7 shows the average message latency of a message for traversing a pipeline chain varying the number of *Map*'s Workers. We consider two configurations: the first one without the *thread-to-core affinity* and the second one with the affinity configured. As in the previous benchmark, for all configurations tested, the total number of system threads is fixed to 64.

The results obtained demonstrate that the two versions perform better than the configuration in which the computing Actor is executed sequentially when the number of *Map*'s Workers is in the range 8–48. Outside this range, there are too few Actors either in the *Map* accelerator to amortize the overhead introduced by the accelerator, or in the CAF run-time system to execute the 800 Actors implementing the benchmark, respectively. Therefore, in such conditions,

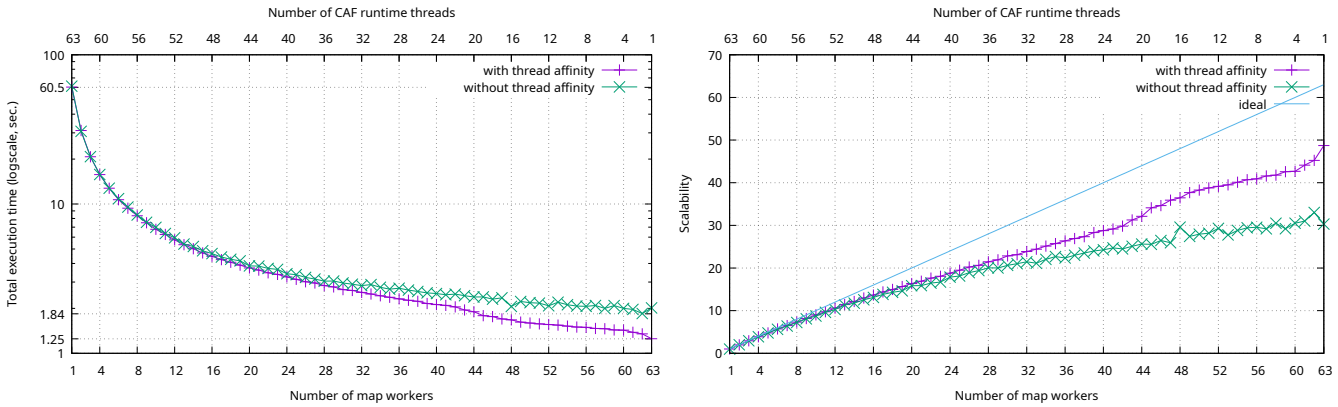

Fig. 4: The execution time (left-hand side) and the scalability (right-hand side) of the data-parallel benchmark

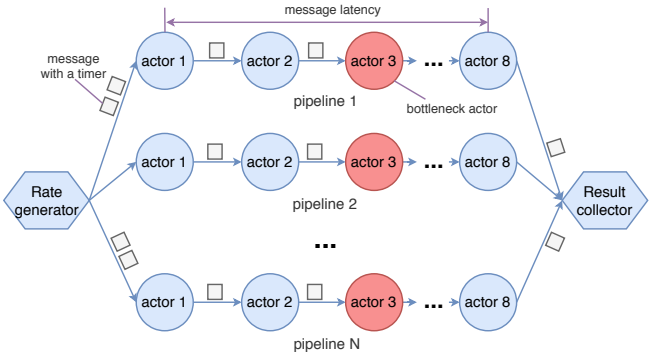

Fig. 5: The CAF Latency benchmark with bottleneck Actors.

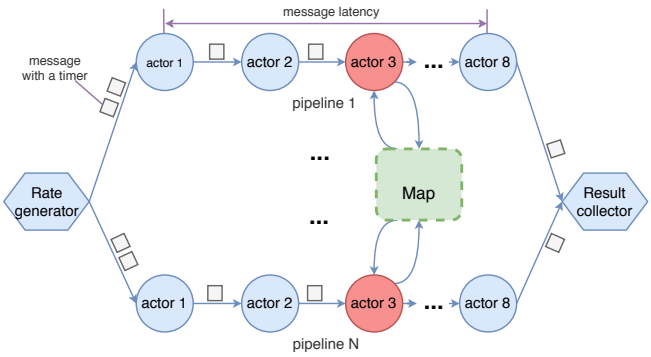

Fig. 6: The *Map* accelerator in the CAF Latency benchmark.

the message latency drastically increases because the two resource sets are not well balanced.

The version with the sequential Actor has an average message latency of about $7ms$, whereas the version with the *Map* accelerator and the *thread-to-core affinity* enabled has an average latency of $0.8ms$. The configuration without *thread-to-core affinity* has an irregular message latency varying the number of Workers. This is due to the interferences of the two set of system threads that the OS scheduler is not always able to evenly distribute to the available core resources.

To conclude, the two benchmarks tested demonstrate that the *Map* accelerator can reduce the service time of Actors performing map-like computations, requiring only a minimal programming effort to the application developer. We do believe that the very same approach can be profitably

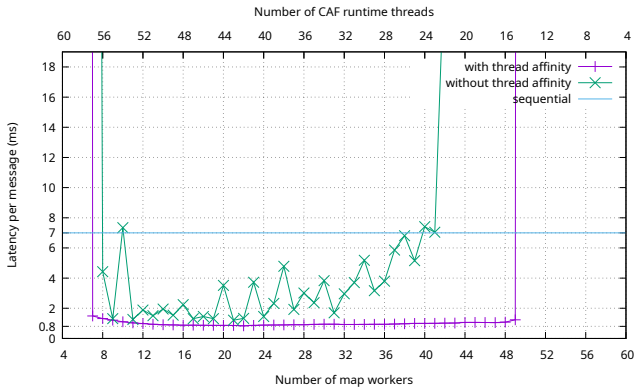

Fig. 7: CAF Latency Benchmark with 100 chains and input vectors of $5,000$ elements.

used also for accelerating the computation of other parallel patterns such as the *pipeline*, the *task-farm* and the *divide-and-conquer*.

V. RELATED WORKS

Here we consider some research works that proposed extensions or improvement of the Actors-based model.

Skel [21] is a parallel library written in Erlang. It provides the user with a set of parallel patterns (e.g., pipeline, farm) that can be composed in a functional way. Each pattern is implemented by using Erlang Actors and can be customized by providing a set of functions. The authors claim that Skel can improve the programmability of the Actor model and can also solve performance degradation issues. Our approach differs from the Skel one. We advocate the use of a software accelerator to seamlessly introduce parallel patterns in the Actor model.

In [22] the authors extends the C++ Actor Framework (CAF) to support external HW accelerator (e.g., GPUs) through OpenCL. The extension implements an OpenCL manager and a new OpenCL Actor. The OpenCL manager supports the interaction with OpenCL capable devices and it can spawn OpenCL Actors. Each OpenCL Actor implements a single OpenCL kernel and thus the spawn operation need the kernel source code, and the definition of the *in* and *out* buffers as parameters. After spawning the OpenCL

Actor, other Actors can send requests of computation that can be forwarded to the defined external accelerator device. Although, the CAF OpenCL extension presents several similarities to our approach, our proposal differ from it in many aspects. First, our approach does not require a specific external hardware accelerator to speed up data-parallel computations. Also, we propose an effective way in which CPU resources can be partitioned between the Actor model and an *Map* accelerator. Moreover, our accelerator supports concurrent interaction with multiple Actors and each of them can execute a different function. Finally, we propose a wider synergy between the Actor model and the parallel patterns approach, where the *Map* accelerator is the first implementation.

The Ray framework [23], is a distributed concurrency framework designed to implement Reinforcement Learning algorithms, capable to cooperate with the most modern Machine Learning libraries. Ray implements a task-based parallel model with *remote procedure calls* and *promises* in combination with some concepts of the Actor model. In particular, Ray implements Actors using the active object model, where Actors are objects with an internal state and a set of exposed methods. The combination of the Actor model with the task-based model is an interesting way to combine stateless pure functions (i.e. tasks) with stateful object with methods (i.e. Ray Actors). Differently from Ray, we aim to bring crafted solutions for common computational problems and to bring a methodology to solve bottleneck problems inside Actors-based applications. Indeed, Ray has a dynamic computational graph, which suffers from the same issue of the Actor model.

In [24] the authors proposed an improved message execution scheduler for the Actor model. They propose a customizable message scheduler that is capable to schedule multiple independent messages at the same time. A simple use-case is that of an Actor cell which receive read and write requests, the actor cell can process multiple read operation concurrently but only one write operation at the same time when no other read operations are processed. From the parallel patterns viewpoint, this problem can be solved by a proper specialization of the *task-farm* pattern. However, the authors propose an AmbientTalk implementation of the scheduler, which uses a thread pool inside the Actor. Our proposal promotes a more general approach to these kind of parallel problems, aiming at providing the programmer with suitable abstractions that can be explicitly instantiated and properly customized to solve the problem at hand.

VI. CONCLUSION AND FUTURE WORK

The Actor model is becoming increasingly popular for implementing parallel applications on multi/many-core systems also thanks to its clear and straightforward programming model and to the capability of managing a large number of concurrent activities. However, the unstructured composition of Actors may easily produce intricated topologies which may hide potential bottleneck that, when discovered, may not be easy to eliminate.

In this paper, we proposed a first attempt to combine the Actor model with the structured parallel programming approach based on parallel patterns by using the concept of software accelerator. In particular, we proposed an accelerator built in the C++ Actor Framework (CAF) that can reduce the service time of Actors implementing data-parallel computations according to the *Map* parallel pattern. The accelerator has been implemented in such a way it does not share resources with the CAF run-time system thus allowing to reduce potential conflicts and to improve resource utilization. To this end, we modify the CAF run-time to be able to control the thread-to-core affinity of system threads.

As a future extension of this work, we planned to implement other parallel patterns such as *Map-Reduce*, *farm*, *flat map*, *divide-and-conquer* and *pipeline* by using the same software accelerator approach. Our primary objective is to facilitate the programmer to compose Actors in a more reliable and structured communication graph. Besides, the accelerator approach deserves further testing, and new benchmarks have to be implemented. We planned to implement one of the reference benchmarks for testing Actor model implementations, namely the SAVINA benchmarks [25]. Finaly, we want to study how non-functional features, such as the number of resource assigned to the accelerator, can be dynamically adapted by using already investigated algorithms and techniques [26], [27].

ACKNOWLEDGMENT

This work was partially supported by ATS S.p.A. with the project “Optimizing mailbox latency and reactivity in CAF”.

REFERENCES

[1] H. Sutter, “A fundamental turn toward concurrency in software: Your free lunch will soon be over. what can you do about it,” *Dr. Dobb’s Journal*, vol. 30, no. 3, pp. 16–22, 2005.

[2] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular actor formalism for artificial intelligence,” in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI’73, Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[3] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, “Orleans: Distributed virtual actors for programmability and scalability,” *Microsoft Research*, Mar. 2014, Available: `https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/`.

[4] J. Armstrong, “The development of Erlang,” in *Proc. of the 2nd ACM SIGPLAN Int. Conference on Functional Programming*, ser. ICFP ’97, Amsterdam, The Netherlands: ACM, 1997, pp. 196–203.

[5] P. Haller and M. Odersky, “Scala actors: Unifying thread-based and event-based programming,” *Theor. Comp. Science*, vol. 410, no. 2, pp. 202–220, 2009.

[6] T. G. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Addison-Wesley Professional, 2004, ISBN: 0321228111.

[7] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto, "Bringing parallel patterns out of the corner: The p3arsec benchmark suite," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, 33:1–33:26, Oct. 2017. DOI: 10.1145/3132710.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[9] A. Ernstsson, L. Li, and C. Kessler, "SkePU2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems," *Int. Journal of Parallel Programming*, vol. 46, no. 1, pp. 62–80, Feb. 2018.

[10] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proc. of the 21th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, Atlanta, Georgia, USA: ACM, 2016, pp. 651–665.

[11] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native actors: A scalable software platform for distributed, heterogeneous environments," in *Proc. of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ser. AGERE! 2013, Indianapolis, Indiana, USA: ACM, 2013, pp. 87–96. DOI: 10.1145/2541329.2541336.

[12] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting actor programming in c++," *Comput. Lang. Syst. Struct.*, vol. 45, no. C, pp. 105–131, Apr. 2016. DOI: 10.1016/j.cl.2016.01.002.

[13] A. Vajda, *Programming many-core chips*. Springer, Boston, MA, 2011, ISBN: 978-1-4419-9738-8.

[14] J. Allen, *Effective akka: Patterns and best practices*. " O'Reilly Media, Inc.", 2013, ISBN: 978-1449360078.

[15] M. McCool, J. Reinders, and A. Robison, *Structured parallel programming: Patterns for efficient computation*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[16] S. Ernsting and H. Kuchen, "Algorithmic skeletons for multi-core, multi-gpu systems and clusters," *Int. J. High Perform. Comput. Netw.*, vol. 7, no. 2, pp. 129–138, Apr. 2012.

[17] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "FastFlow: High-Level and Efficient Streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds., accepted in 2012, John Wiley & Sons, Inc, Jan. 2017, ch. 13. DOI: 10.1002/9781119332015.ch13.

[18] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, 134:1–134:25, Apr. 2014. DOI: 10.1145/2584665.

[19] D. del Rio Astorga, M. F. Dolz, J. Fernandez, and J. D. Garcia, "A generic parallel pattern interface for stream and data processing," *Concurrency and Computation: Practice and Experience*, 2017, e4175 cpe.4175.

[20] M. Torquati, T. Menga, T. D. Matteis, D. D. Sensi, and G. Mencagli, "Reducing Message Latency and CPU Utilization in the CAF Actor Framework," in *2018 26th Euromicro Int. Conference on Parallel, Distributed and Network-based Processing (PDP)*, Mar. 2018, pp. 145–153. DOI: 10.1109/PDP2018.2018.00028.

[21] I. Bozó, V. Fordós, Z. Horvath, M. Tóth, D. Horpácsi, T. Kozsik, J. Köszegi, A. Barwell, C. Brown, and K. Hammond, "Discovering parallel pattern candidates in erlang," in *Proceedings of the 13th ACM SIGPLAN Workshop on Erlang*, ser. Erlang '14, Gothenburg, Sweden: ACM, 2014, pp. 13–23. DOI: 10.1145/2633448.2633453.

[22] R. Hiesgen, D. Charousset, and T. C. Schmidt, "OpenCL Actors – Adding Data Parallelism to Actor-Based Programming with CAF," in *Programming with Actors: State-of-the-Art and Research Perspectives*, A. Ricci and P. Haller, Eds. Cham: Springer International Publishing, 2018, pp. 59–93. DOI: 10.1007/978-3-030-00302-9_3.

[23] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," *ArXiv preprint arXiv:1712.05889*, 2017.

[24] C. Scholliers, É. Tanter, and W. De Meuter, "Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model," *Science of Computer Programming*, vol. 80, pp. 52–64, 2014. DOI: 10.1016/j.scico.2013.03.011.

[25] S. M. Imam and V. Sarkar, "Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries," in *Proceedings of the 4th Int. Workshop on Programming Based on Actors Agents & Decentralized Control*, ser. AGERE! '14, Portland, Oregon, USA: ACM, 2014, pp. 67–80. DOI: 10.1145/2687357.2687368.

[26] M. Aldinucci, M. Danelutto, and P. Kilpatrick, "Autonomic management of non-functional concerns in distributed amp; parallel application programming," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.

[27] D. De Sensi, M. Torquati, and M. Danelutto, "A reconfiguration algorithm for power-aware parallel applications," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, 43:1–43:25, Dec. 2016, ISSN: 1544-3566.