Noname manuscript No.
(will be inserted by the editor)

Improving the performance of Actors on Multi-Cores with Parallel Patterns

Luca Rinaldi · Massimo Torquati* · Daniele De Sensi · Gabriele Mencagli · Marco Danelutto

Received: date / Accepted: date

Abstract The Actor-based programming model is largely used in the context of distributed systems for its message-passing semantics and neat separation between the concurrency model and the underlying hardware platform. However, in the context of a single multi-core node where the performance metric is the primary optimization objective, the "pure" Actor Model is generally not used because Actors cannot exploit the physical shared-memory, thus reducing the optimization options. In this work, we propose to enrich the Actor Model with some well-known Parallel Patterns to face the performance issues of using the "pure" Actor Model on a single multi-core platform. In the experimental study, conducted on two different multi-core systems by using the C++ Actor Framework (CAF), we considered a subset of the PARSEC benchmarks and two SAVINA benchmarks. The analysis of results demonstrates that the Actor Model enriched with suitable Parallel Patterns implementations provides a robust abstraction layer capable of delivering performance results comparable with those of thread-based libraries (i.e. PTHREADS and FASTFLOW) while offering a safer and versatile programming environment.

Keywords Actors · Parallel Patterns · Programming Model · Multi-Cores

1 Introduction

The Actor Model (AM) proposed by Hewitt et al. in 1973 [24] is attracting a revived attention among software developers and academics. In the AM, the

* Corresponding author – E-mail: torquati@di.unipi.it

This work has been partially supported by Univ. of Pisa PRA 2018 66 DECLware: Declarative methodologies for designing and deploying applications

Luca Rinaldi · Massimo Torquati · Daniele De Sensi · Gabriele Mencagli · Marco Danelutto
Computer Science Department
University of Pisa, Italy

concurrent unit is the *Actor*. Actors are isolated entities with an internal state that can receive messages from other Actors, perform computations, manage their internal state, and send messages to other Actors. An Actor can interact with other Actors only if it knows their logical addresses. Two distinct Actors do not share any data, so low-level data races are avoided by design. The model allows a complete separation of the software design from its deployment at runtime. Notable implementations of the AM are: Akka [4] written in Java/Scala, Orleans [7] written in C#/.NET, Erlang [5] with its specific language, and the "C++ Actor Framework" (CAF) entirely written in modern C++ [13]. The message-passing style of non-blocking and asynchronous interactions via immutable messages among Actors, make the AM particularly attractive for exploiting the potential parallelism of large distributed systems to target scale-out scenarios. However, in the context of a single multi-core node where the application programmer wants to maximize the performance metric of his/her Actor-based application (i.e. scale-up scenario), the AM is not largely used because of its performance issues related to the narrow margins offered by the AM to exploit the physical shared memory of the system. In a way, the classical (or "pure") AM trades single node performance for scalability to a large number of nodes.

Several research efforts tried to overcome the performance issues of the "pure" AM on the single node by implementing smart techniques at the Run-Time System (RTS) level, e.g., copy-on-write and locality-aware scheduling [20]. However, none of these specific and low-level optimizations were capable of significantly increasing the performance of Actor-based applications on shared-memory platforms at the same level of concurrent libraries designed explicitly for multi-cores, such as OpenMP, Intel TBB [31], and FastFlow [3].

Some popular library-based implementations of the AM are built on top of languages that allow shared-memory visibility. Examples are Akka using Java, and CAF using modern C++. Such implementations cannot enforce the *Actor isolation* property [16] because the programmer can always define some global state that can be shared by some of the Actors. On the one hand, this gives high flexibility to the programmer allowing him/her to use the physical shared-memory for tuning and optimizing performance-critical parts of the Actor-based application. On the other hand, the usage of the shared-memory exposes the user to potential data races and data inconsistencies that have to be explicitly handled by the application programmer through proper synchronizations (either via message exchange or by using locks), which partially vanishes the programmability benefits of using the "pure" model.

In this work, we aim to increase the performance of the "pure" AM on multi-cores by leveraging the physical shared-memory without compromising the Actor's isolation property and avoiding explicit management of the shared-memory by the application programmers. The idea is to provide the Actor programmer with a set of *Parallel Patterns* (PPs) [30], i.e. high-level parallel programming abstractions, with a well-defined functional and parallel semantics that can be used to parallelize a given problem (or a given class of problems). Examples of PPs are: *Pipeline*, *Map-Reduce*, *Task-Farm*, and

Stencil. To fully integrate PPs in the AM, we propose to implement them as "macro Actors" so that they look like standard Actors to the programmer. They interact with other Actors and PPs only by using explicit messages, and they can be dynamically spawned. The shared-memory and all low-level platform-specific optimizations, such as reference passing to avoid data copies, native threads exploitation, and thread-to-cores affinity [32], are used only in the *implementation skeleton* of the PP and they are entirely transparent to the application programmer. Therefore, a PP behaves like a standard Actor, but its internal implementation, based on concurrent Actors, does not necessarily comply to the "pure" AM. Moreover, since the implementation schemes of PPs are provided to the user as a library with suitable APIs, this enables the *separation of concerns* software design principle between the application programmer and the patterns provider. The application programmer has the responsibility to select the proper PP, whereas the patterns provider has the responsibility to produce an efficient and memory-safe implementation of PPs.

We implemented PPs as an open-source library for the "C++ Actor Framework" [12] (CAF). We validate our approach over a set of applications chosen according to the computational model that the PPs implement (e.g., Divide&Conquer, Map, Pipeline, Farm). In particular, we consider *QuickSort* and *Recursive Matrix Multiplication* from the SAVINA benchmark suite and `blackscholes`, `ferret`, `canneal`, and `raytrace` from the PARSEC benchmark suite. The results obtained by running the benchmarks on two different multi-core platforms (Intel Xeon and IBM Power8), demonstrate that the AM enriched with a proper set of PPs provides an enhanced programming model capable of delivering performance results comparable with those obtained by thread-based libraries (i.e. PTHREADS and FASTFLOW [3]) on multi-cores while offering a safer and more flexible programming environment.

The remaining of this paper is organized as follows: Section 2 provides the background information. Section 3 describes the design and implementation of the PPs, Section 4 describes and discusses the results obtained during the experimental phase. Section 5 presents related works and Section 6 summarizes the main results.

2 Background

In this section, we provide the necessary background to understand the contributions of this work. Specifically, we summarize the main concepts of the Actor Model and of the Parallel Pattern approach to parallel programming. Finally, we provide an overview of the CAF framework that we used for implementing the PPs presented in Section 3.

The Actor Model. The *Actor Model* is a concurrent programming model first proposed by Hewitt et al. [24] in the context of Artificial Intelligence to model systems with thousands of independent processors, each having a local memory and connected through some high-performance network. Later, the AM has been formalized by Agha et al. [1, 2]. In the AM, every distinct

execution flow is considered an Actor. Actors are uniquely identified by an opaque identifier (*address*) so that they can be addressed during send operations. Actors can dynamically spawn other Actors. They do not share data, and the only way to observe or modify the internal state of an Actor is through explicit messages. Each Actor has a private mailbox of unbounded capacity, which stores an ordered set of messages received by that Actor. However, there is no guarantees on the processing order of messages. The messages can be sent to any other Actor that processes them asynchronously and sequentially. The *memory isolation*, the *message-passing style of communication*, and the *serial execution of messages*, allow to eliminate the use of locks when the AM is employed on shared-memory systems. Synchronization is achieved only through the exchange of messages. In principle, in this model, the programmer is not aware of the underlying platform, therefore the Run-Time System (RTS) is the only responsible for the efficient execution of Actors.

Pattern-based parallel programming. One of the approaches for raising the level of abstraction in parallel computing is based on the concepts of *Parallel Patterns* [30] and *Algorithmic Skeletons* [15], which are schemes of parallel computations that recur in many applications and algorithms. 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 (often called *implementation skeleton*) on the basis of the parallel platform considered. Different models of communications synchronizations and coordination for the tasks execution can be used to implement a given pattern. Notable examples of PPs are *Pipeline*, *Map*, *Map-Reduce*, *Task-Farm*, *Divide&Conquer*. Combinations of PPs and Algorithmic Skeletons are used in several programming frameworks and libraries such as Microsoft PPL [11], Intel TBB [31], SkEPU [18] and FastFlow [3].

The CAF library. The C++ Actor Framework (CAF) is an Actor-based framework implemented in modern C++ [12,13]. Actors are modeled as lightweight state machines that are mapped onto a fixed set of RTS threads called *Workers*. Instead of assigning dedicated threads to Actors, the CAF RTS includes a scheduler (which implements a *Work-Stealing* algorithm) 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.

In CAF, Actors are created using the *spawn* function, which can create an Actor from a function, or a class. 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 a set of *behaviors* each of which is a C++ lambda function.

Actors using blocking system calls (e.g., I/O functions) might suspend a RTS thread creating either imbalance in the workload or starvation. To solve

these issues, CAF provides *detached Actors* that will be executed by a dedicated OS thread, instead of the Work-Stealing scheduler. Although detached Actors have their private executors, they implement the same event-based protocol for the execution of mailbox messages as the default lightweight Actors (the default Actors are also called *scheduled Actors*). When a *detached Actor* is spawned, a new OS thread is created. This increases the management costs of these Actors if compared with *scheduled Actors*, but it enables more flexibility and full control of the Actor scheduling and of its processor affinity [32].

3 Designing Parallel Patterns as Actors

Actor-based applications are characterized by unstructured communication graphs where Actors are often created dynamically and have a short life-span. Besides, it has been observed that, in many Actors-based applications, a small number of Actors (called *hub Actors*) exchange significantly more messages than the other ones composing the application [20].

By means of PPs abstractions, we want to bring a communication structure in Actor-based applications, and to enable some optimizations, which are generally not allowed by the "pure" AM without breaking the principles of the model itself (e.g., sharing a data structure by exploiting the physical shared-memory). During the design of PPs, we used the following guidelines:

- PPs must smoothly integrate with existing Actors;
- PPs interface must comply with the AM;
- to maximize the performance, the implementation skeletons of PPs could not necessarily respect the AM, and they can rely on all low-level features and mechanisms offered by the shared-memory platform.

We designed the PPs in such a way that they look like standard Actors, i.e., they receive input data only through messages and produce results by sending messages to other Actors. Instead, their implementation skeletons use the shared-memory concurrency model. We decided to implement these PPs by leveraging the Actors provided by the CAF library without introducing another model/library (i.e., OpenMP or FastFlow) to avoid issues of mixing different RTSs (e.g., defining the number of threads of each RTS, handling different affinity policies, handling different scheduling of tasks). Therefore, a pattern in the PPs library is implemented by spawning a set of CAF Actors cooperating in a predefined communication scheme through explicit messages and shared memory variables. Moreover, CAF offers the option of spawning Actors also as private threads (i.e. *detached Actors*), thus enabling the possibility to control Actors directly without using the Work-Stealing scheduler (which is used instead for *scheduled Actors*). This permits to avoid the indirection between the logical Actor entity and the underline RTS threads used to execute Actors, which, sometimes, may introduce extra overhead.

In the following, we discuss the implementation of a set of PPs[1] implemented on top of the CAF framework, dividing them in two sets: *Data-*

[1] The implementations are available at `https://github.com/ParaGroup/caf-pp`.

parallel PPs namely *Map*, *Divide&Conquer*; and *Control-parallel PPs* namely *Seq*, *Pipeline*, and *Farm*. The first set of patterns internally exploits shared-memory to optimize the performance. The second set enables nesting and composition of PPs focusing more on structuring the concurrent graph of Actors and PPs.

3.1 Data-parallel PPs

Map. The *Map* pattern is a data-parallel paradigm that applies the same function to every element of an input collection. The input collection of data, possibly but not necessarily coming from a stream of collections, is split into multiple sub-collections where each one can be computed in parallel by applying the same function. The results produced are collected in one single output collection, usually having the same type and size of the input.

The efficiency of the Map pattern on multi-cores depends on the ability to share the input collection on which the user function has to be applied. Data races are avoided by design because the parallel semantics of the Map pattern is such that distinct concurrent entities work on disjoint sub-collections. Data sharing has the advantage of avoiding costly data copies required by the message-passing model.

Figure 1 shows the implementation skeleton of the *Map* pattern. It uses a "master" entity, called `Sched`, which is in charge of partitioning the input collection and scheduling data partitions toward a pool of Worker entities. The `Sched` also waits for the end of the computation of the Workers to implement a barrier before sending out the final result.

Listing 1 shows how to configure and spawn a *Map* pattern with the PP library. The user provides a C++ lambda function that works in-place on a specific range of elements of the input collection implemented with a container. Both the number of internal Workers and the scheduling policy are two optional parameters. The first, if not set, is equal to the number of active cores; the second parameter can be set to either `static` assignment of partitions (the default value) or `dynamic` assignment of partitions.

The static scheduling policy splits the input collection into several partitions equal to the number of Workers. The `Sched` Actor sends the references of each partition to the corresponding Worker. This policy works well when the computational workload is equally (or almost equally) distributed among all elements of the input collection. The dynamic scheduling policy, instead, is supposed to be used when a static partitioning of the input collection may lead to serious workload balancing issues among Workers.

The dynamic policy gets as argument a user-defined chunk size used to split the input collections. Then, chunks of data elements are dynamically fetched by the Workers leveraging the C++ *std::atomic* data type implementing an *atomic counter* shared by all Workers and initialized by the `Sched` Actor. It is worth to remark that the shared atomic counter is used only to implement the scheduling policy, and it is visible only to the Worker Actors implementing the *Map* pattern, which are not defined by the application programmer.

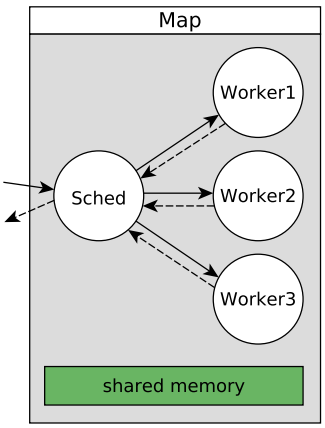


Fig. 1: The *Map* pattern implementation scheme.

```
using Cnt = std::vector<int64_t>;
auto map = Map<Cnt>([](auto range) {
    for (int64_t &el : range) {
        // do somethings with el
    }
}).replicas(3)
  .scheduler(PartitionSched::static_())
//.scheduler(PartitionSched::dynamic_{1})
  .runtime(Runtime::actors);
//.runtime(Runtime::threads);
auto p = spawn_pattern(sys, map).value();
```

Listing 1: The code to build and spawn the *Map* pattern.

The `Sched` Actor initializes the atomic counter to zero and sends a first message to all Workers containing the size of the collection and the number of elements to fetch each time the shared variable is accessed (i.e. the computation granularity). Each Worker executes a `fetch_add` atomic operation on the shared variable to retrieve the next range of contiguous collection elements to compute. The computation finishes when all Actors retrieve a range of collection elements whose iterator indexes are greater than or equal to the number of elements in the collection. Then, the Workers notify the `Sched` Actor of their work completion by sending an appropriate message.

CAF enforces the Actor isolation property by calling the C++ copy constructor on those message objects sent to more than one destination Actors, which do not use the input message in read-only mode (i.e. non-constant input data types). To work in-place (i.e. in a read-write mode) on message types that are input collections, the *Map* PP inhibits those implicit copies to enable the sharing of the same collection to multiple Workers. To this end, we defined a C++ type, called `NsType`, which internally manages a heap-allocated data and implements the copy constructor without effectively doing a memory copy. The `Sched` Actor moves the user input collection inside a `NsType` object, and then sent it to the Workers. After the computation, the `Sched` Actor executes the same steps in the reverse order and produce the output result. This implementation guarantees that the shared-memory layer inside the PP is transparent to the application programmer.

Divide&Conquer In Divide and Conquer (D&C) algorithms, during the Divide (or Split) phase, the problem is recursively decomposed into smaller sub-problems building a tree of calls. In the Conquer (or Merge) phase, the partial results produced by the solution of the sub-problems at a given level of the tree are adequately combined to build the final result. A D&C algorithm can be parallelized by executing, on different CPU cores, the Split and Merge phases

for those sub-problems that do not have a direct dependency in the recursion tree. At each level of the tree, a new set of concurrent tasks is available to be executed up to the point where the sub-problems are small enough that it is more convenient to compute them using the sequential algorithm.

As shown in Figure 2, we implemented this pattern by dynamically spawning CAF Actors, which recursively spawn new Actors for each sub-problem produced by the `divide` function. The Actor spawned evaluates the *condition* function. If this function returns false, the *divide* function is called. If it evaluates to true (e.g., when the size of the sub-problem is smaller than a given *cut-off* value), the *sequential* algorithm is called, and the partial result produced is returned back to the spawning Actor. The generic spawning Actor waits for all partial results, and then it executes the `merge` function whose result will be sent to its spawning Actor, and then it terminates. The *DivConq* implementation skeleton uses the physical shared-memory to avoid unnecessary data copies during the dynamic spawning of Actors, which work in-place on different input ranges by using the same techniques described for the *Map*.

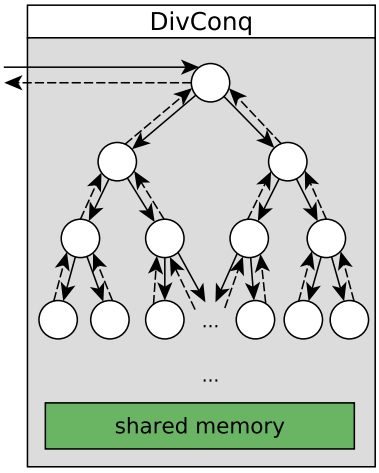


Fig. 2: The *DivConq* pattern implementation scheme.

```
1  auto div = [](Rng&)-> vector<Rng> {
2    // divide the input
3  };
4  auto merg = [](vector<Rng>&)-> Rng {
5    // merge the partial results
6  };
7  auto seq = [](Rng&) {
8    // base case
9  };
10 auto cond = [cutoff](const Rng&)-> bool {
11   // splitting condition
12 };
13 DivConq<Cnt> dc(div, merg, seq, cond);
14 auto p = spawn_pattern(sys, dc).value();
```

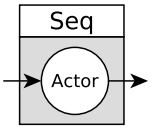
Listing 2: The code to build and spawn the *DivConq* pattern.

Listing 2 shows the interface of the *DivConq* PP. The pattern is created by passing a user defined *Container* and four functions that work on continuous portions of the input container called *Ranges* (`Rng` in Listing 2).

3.2 Control-parallel PPs

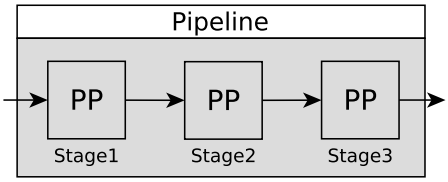
Seq. The *Seq* (Sequential) represents a single concurrent entity and it is useful to integrate within the PPs library a custom Actor implemented by the user.

The *Seq* pattern can be seen as a factory of a specific CAF Actor. It allows to spawn different copies of the same Actors and to use them in different points of a PPs composition. In the left-hand side of Figure 3 there is the implementation scheme of the *Seq* pattern. The right-hand side of Figure 3 shows two ways of creating a *Seq* pattern from an existing Actor type `MyAct`. In the first case (line 1) the Actor will be initialized without any parameter. In the second case (line 2), the user provides a callback that will be called as soon as the user Actor will be spawned to initialize it.



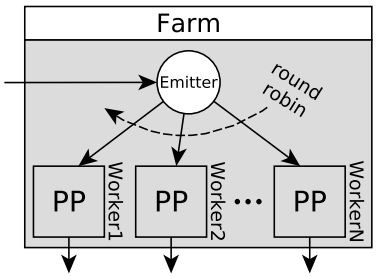

```
1  Seq<MyAct> seq1; // with no initialization
2  Seq<MyAct> seq2([&](caf::actor a)
3    { caf::anon_send(a, par1, par2); }); // with initialization
```

Fig. 3: The *Seq* pattern implementation scheme (right). The example code for building a *Seq* by using the `MyAct` CAF Actor (left).




```
1  Seq<MyAct1> s1;
2  Map<Cnt>([](auto range){/*...*/}) s2;
3  Seq<MyAct2> s3;
4  Pipeline pipe(s1, s2, s3);
5  auto p = spawn_pattern(sys, pipe)
6           .value();
```

Fig. 4: The *Pipeline* pattern implementation schema (top). An example code for building and spawning an instance of a three-stage pipeline (bottom).




```
1  Seq<Worker> worker;
2  auto farm = Farm(worker).replicas(N)
3              .policy(round_robin());
```

Fig. 5: The *Farm* pattern implementation schema (top). An example code for building a *Farm* pattern with N sequential Workers and the *round-robin* policy (bottom).

Pipeline. A pipeline pattern is a sequence of stages connected in a linear chain. Distinct stages of the same chain work in parallel on subsequent input elements (usually called *stream of items*). Each stage computes a partial result and sends it to the next stage of the sequence. The stages of the *Pipeline* pattern can be any sequence of PPs presented in this section, as shown in Figure 4. The *Pipeline* takes care to connect each stage in the correct order.

Farm. A Farm pattern is composed of a pool of concurrent entities called Workers executing in parallel on different data elements of the input stream. Input elements are forwarded to the Workers according to some predefined

scheduling policy (e.g., round-robin, random, etc.), or using a user-defined policy. Precisely, the *Farm* pattern replicates a given number of times the PP provided as an argument. As for the *Pipeline*, the argument can be any valid combination of the patterns presented in this section. The number of replicas can be left unspecified, meaning that a default value will be used (e.g., the number of active CPU cores). Finally, it is also possible to customize the scheduling policy for the input messages by defining a function with a specific signature. Figure 5 shows both the internal implementation scheme of the *Farm* pattern and a simple example code for instantiating it.

3.3 Composition of Parallel Patterns

The patterns in the PPs library, can be composed to build more complex computation structures. Specifically, the *Farm* and *Pipeline* patterns can have as internal elements any patterns, while *Seq*, *Map* and *Divide&Conquer* are leaf-nodes of the skeleton tree composition and they cannot contain other patterns.

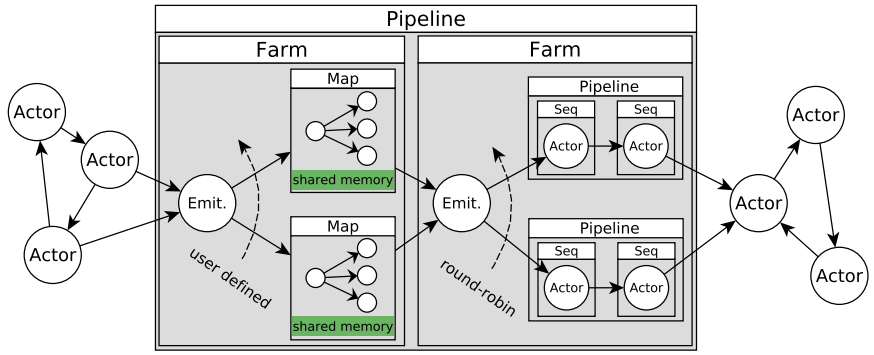


Fig. 6: Composition of two *Farms* using a *Pipeline* pattern. The first *Farm* has a *Map* pattern replica as Worker, whereas the second *Farm* uses *Pipeline* of *Seq* as Workers.

Figure 6 exemplifies how a pattern composition can be used inside of an Actor-based application. The *Pipeline* pattern is fed by two standard Actors, and the results produced by the Workers of the second *Farm* pattern (i.e. by the last stage of each *Pipeline* Workers) are sent to another standard Actor of the application through messages.

4 Evaluation

The experiments were conducted on two different multi-cores (`Xeon` and `Power8`) considering a subset of SAVINA [26], and PARSEC [8] benchmarks.

Xeon. A dual-socket Intel E5-2695 Ivy Bridge CPUs running at 2.40GHz and featuring 24 cores (12 per socket). Each hyper-threaded core has 32 KB private L1, 256 KB private L2 and 30 MB of L3 shared cache. The machine has 64 GB of DDR3 RAM, using Linux 3.14.49 x86_64 with the CPUfreq performance governor enabled. Available compiler `gcc` version `7.2.0`.

Power8. A dual-socket IBM server 8247-42L with two Power8 processors each with ten cores organized in two CMPs of 5 cores working at 3.69GHz. Each core (8-way SMT) has private L1d and L2 caches of 64 KB and 512 KB, and a shared on-chip L3 cache of 8 MB per core. The total number of cores is 20 physical and 160 logical. The machine has 64 GB of RAM, using Linux 4.4.0-47 ppc64 shipped with Ubuntu 16.04. Available compiler `gcc` version `8.2.0`.

Savina. It is a set of benchmarks specifically conceived for evaluating AM implementations. They can be classified in three categories: i) micro-benchmarks, ii) concurrency benchmarks, and iii) parallelism benchmarks. The first set contains simple benchmarks dedicated to test specific features of the Actor RTS (e.g., time to spawn an Actor). The second set contains classical concurrency problems (e.g., Dining-Philosophers). The third set includes applications that demand more computation (e.g., Matrix Multiplication). We selected two applications of the *parallelism benchmarks* set, namely `quicksort` and `recMM`[2], because they are both implemented using recursive algorithms (which are not present in the Parsec benchmarks), and because they are well-known problems with a straightforward implementation in the "pure" Actor Model.

Parsec. It is a collection of several complex parallel applications for shared-memory architectures with high system requirements. Indeed, they are real applications covering many different domains such as streaming applications, scientific computing, computer vision, data compression and so forth. Recently, the Parsec benchmarks have been used to compare and assess programming models targeting multi-cores [14, 17]. For testing the PPs library , we selected `ferret`, `blackscholes`, `raytrace` and `canneal` benchmarks[3]. The first one is a data streaming application, whereas `blackscholes` and `raytrace` are two data-parallel applications with different computational granularity and different workload balancing issues. The last one is a fine-grained master-worker computation.

All experiments have been executed multiple times and the average value obtained has been used to compute the speedup reported in the folloowing plots. The standard deviation is generally low and not reported in the plots.

In the next two subsections, we first highlight the performance problems of using the "pure" AM on multi-core platforms and how the PPs proposed can be used to significantly improve the performance without breaking the model. Then, by using the Parsec benchmarks, we compare the performance of the PPs library with that obtained by using the native Pthreads implementation shipped with Parsec, and the FastFlow implementation that uses the same

[2] Application code available at `https://github.com/ParaGroup/caf-pp`

[3] Application code available in the P[3]ARSEC repository at `https://github.com/ParaGroup/p3arsec`

PPs approach to parallelize the benchmarks. It is worth to remark that the FastFlow performance of Parsec benchmarks has been already compared with other specialized frameworks on multi-cores demonstrating comparable (and in some cases better) overall performance [17].

Fig. 7: Improvement of the PPs version compared to the "pure" AM version of the `quicksort` benchmark on the `Xeon` and `Power8` platforms.

4.1 "Pure" Actor Model vs Actors+Parallel Patterns

Here we compare the performance of the "pure" AM with the AM enriched with the PPs library. We consider `quicksort` and `recMM` from Savina benchmarks, and `blackscholes` from the Parsec benchmark suite. The `quicksort` application implemented in the Savina benchmark follows the "pure" AM semantics. There are not shared variables among Actors. During the Split and the Merge phases of the recursive algorithm, the sub-vectors are copied both before sending them to the spawned Actors and when the results come back. Instead, in the *DivConq* pattern implementation, the internal pattern shared-memory is used to work in-place on the original input vector avoiding unnecessary copies.

Figure 7 shows the performance improvement of the PPs approach with respect to the "pure" AM implementation of the `quicksort`, considering different size of the input vector, i.e. 10M, 50M, and 100M elements, and a fixed cut-off value of $2,000$ elements. As expected the performance improvement increases with the vector size, because of the overhead of copying message data in the Savina implementation. The two versions have roughly the same maximum scalability (namely $\sim$3.5 on the `Xeon` and $\sim$4.2 on the `Power8`), but very different maximum speedup (3.7 vs 1.0 on the `Xeon`, and 4.2 vs 1.0 on the `Power8`, for the PPs version vs the "pure" AM implementation, respectively).

Differently from the `quicksort` application, the `recMM` implementation in the Savina benchmarks does not use a "pure" AM implementation. In fact, all Actors share both the two input matrices as well as the resulting matrix. In this case, messages are used as a synchronization mechanism for accessing the shared data. We implemented the application using the *DivConq* PP which

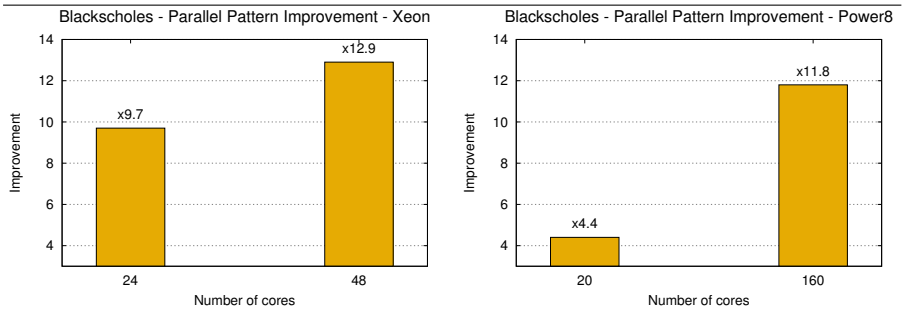


Fig. 8: Improvement of the PPs version compared to the "pure" AM version of the `blackscholes` benchmark on the `Xeon` and `Power8` platforms.

allows us to confine in a cleaner way the Actors that share the data. As expected, the patterned version and the SAVINA version perform almost the same (both obtain a maximum speedup of more than 22 on both platforms, by using matrices of $4,096 \times 4,096$ elements and a cut-off value of 128×128). Then, we implemented the SAVINA version without any data sharing in a "pure" AM fashion. The results obtained (not reported here for space reasons) show an improvement of the PPs version of about five times on the `Xeon` and of about six times on the `Power8`. The scalability of the two versions are roughly the same ($\sim$18 on the `Xeon` and $\sim$21 on the `Power8`), whereas the maximum speedup of the PPs version is much higher on both platforms. These results confirm the importance of exploiting the physical shared memory to optimize the performance metric of Actor-based applications on multi-cores.

Lastly, we consider the `blackscholes` PARSEC benchmark, a real application that prices a portfolio of European options using the Black-Scholes partial differential equations [9]. This application can be parallelized iterating a fixed number of times a *Map* pattern [17]. We first implemented a "pure" Actor version of the *Map* parallel pattern. Then, we used the *Map* pattern presented in Section 3 for producing a second version. In the first version, we used a feature of the CAF library that allows to share the same input message if it is used in read-only mode by the receiving Actors. With this feature, a master Actor sends the input container to a pool of Worker Actors. Then, each Actor internally creates a new output vector for storing the partial result. The partial results are then collected by the master Actor, which merges them producing the final result. This implementation creates two copies of the input vector at each iteration, where one is created in parallel by the Worker Actors. Although this implementation already uses an optimization of the "pure" AM, it performs considerably worse than the one based on the *Map* PP, which internally makes more extensive use of the shared memory (Figure 8). In the next subsection, together with other benchmarks, we compare the speedup of the PP-based `blackscholes` application against other parallel frameworks.

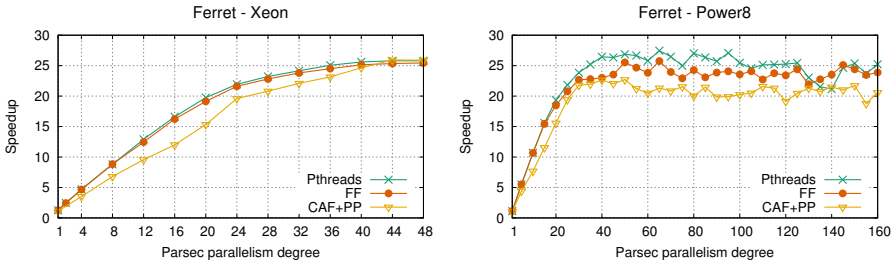


Fig. 9: Speedup of the `ferret` benchmark of the PTHREADS, FASTFLOW (FF) and CAF plus PPs (CAF+PP) implementations on the `Xeon` and `Power8` platforms considering as baseline the PTHREADS version with 1 thread.

4.2 Testing Actors+Parallel Pattern with some PARSEC benchmarks

In this section, we study the PPs-based parallelization of the PARSEC benchmarks selected. The performance results obtained are compared with those of the native PTHREADS and FASTFLOW implementations. The FASTFLOW implementation uses the same pattern-based approach of the PPs library, as described in De Sensi et al. [17].

ferret. This application is based on the Ferret toolkit [29] used for content-based similarity search on different kinds of data, including images, audio and video. In the PARSEC benchmark, the toolkit is configured to perform similarity search on images. In the PTHREADS implementation, the application is composed by six different stages. The first and last stages are sequential while each of the other stages is executed by a separate thread pool. Different pools communicate by using fixed-size queues. The implementation uses a single *Pipeline* pattern containing four *Farm* patterns as stages, each one with the same number of Worker Actors (implemented with *Seq* patterns). The first and last stages of the pipeline are instead *Seq* patterns reading and writing from/to the local disk. The same logical nesting of PPs is used in the FASTFLOW version. Figure 9 shows the speedup of the `ferret` benchmark on the two platforms considered. The results obtained by using the PPs library are comparable to those obtained by both the native PTHREADS and FASTFLOW implementations.

blackscholes and **raytrace**. The CAF implementations of `blackscholes` and `raytrace` use the *Map* pattern described in Section 3. `blackscholes` applies the same function to all elements of an array. The computation is repeated a fixed number of times (100 in our test). `raytrace` implements a computation over an input matrix representing, at different time intervals, a frame of an animated scene. The primary difference between these two data-parallel computations is that `raytrace` has a very unbalanced workload both within the single frame as well as between different frames. Instead, for `blackscholes`, the workload is almost evenly distributed among all elements of the array. Therefore, for `blackscholes` it is reasonable to use a static scheduling policy of the array's partitions while for `raytrace` a dynamic scheduling policy of the frame's partitions has to be used to obtain acceptable speedup.

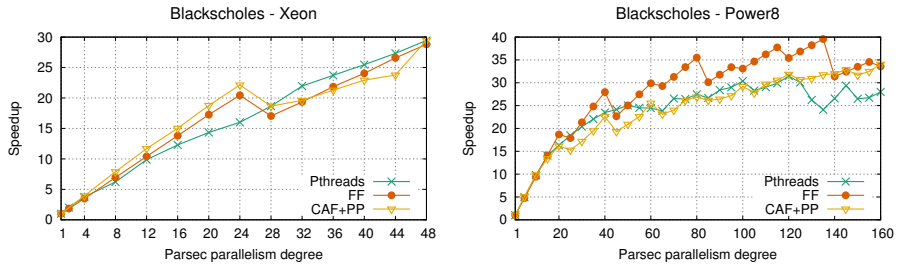


Fig. 10: Speedup of the `blackscholes` benchmark of the PTHREADS,FASTFLOW (FF) and CAF plus PPs (CAF+PP) implementations on the `Xeon` and `Power8` platforms considering as baseline the PTHREADS version with 1 thread.

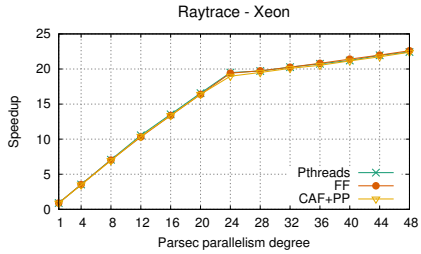


Fig. 11: Speedup of the `raytrace` benchmark of the PTHREADS, FASTFLOW (FF) and CAF plus PPs (CAF+PP) implementations on the `Xeon` platform considering as baseline the PTHREADS version with 1 thread.

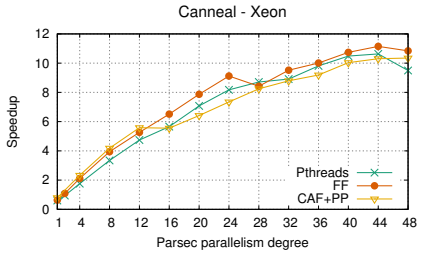


Fig. 12: Speedup of the `canneal` benchmark of the PTHREADS, FASTFLOW (FF) and CAF plus PPs (CAF+PP) implementations on the `Xeon` platform considering as baseline the PTHREADS version with 1 thread.

Figure 10 shows the speedup of the `blackscholes` benchmark. The speedup of the PPs version is close to the other two versions on the `Xeon` platform. On the `Power8` platform the speedup is aligned with that of the native PTHREADS implementation. After every iteration, the `Sched` Actor waits for the completion of all Workers before sending back the final result to the spawning Actor, which then starts a new iteration on the same array (barrier synchronization). The barrier is implemented by using standard inter-Actor messages. During the tests, we found that the barrier synchronization between Actors takes less time if the entire pattern is spawned as detached (cf. Section 3).

The `raytrace` benchmark parallelization has been implemented with the *Map* pattern. Differently from `blackscholes`, it uses the dynamic scheduling policy with a chunk size of 1 element. Figure 11 shows the speedup of the `raytrace` benchmark on the `Xeon` platform (this benchmark does not compile on the `Power8` platform due to some assembly instructions used in the PARSEC implementation). In this case, the CAF version performs almost identically to the PTHREADS and the FASTFLOW versions, confirming the low-overhead introduced by the implementation skeleton of the *Map* pattern.

canneal. This application minimizes the routing cost of a chip design. The algorithm applies random swaps between nodes and evaluates the cost of the new configuration. If the new configuration increases the routing cost, the algo-

rithm performs a rollback step by swapping the elements back. The PTHREADS version follows an unstructured interaction model among threads that execute atomic instructions on shared data structures. At the end of each iteration, a barrier is executed and each thread checks the termination condition. The FASTFLOW implementation instead, uses a master-worker pattern in which the master evaluates the termination condition and restarts the Workers if the termination condition is not met. We implemented the same logical schema used in the FASTFLOW version by using a standard CAF Actor connected to the *Map* PP. The CAF Actor is the master Actor that checks the termination condition, whereas the *Map* pattern is used for the computation as a "software accelerator" (i.e., the result of the computation is sent back to the master Actor). The *Map* pattern uses a static scheduling policy, and the input container has as many entries as the number of Worker Actors so that each Worker works on a single element of the container. If the termination condition is met on the result produced by the *Map*, the master Actor stops the computation. Otherwise, the process is repeated.

Figure 12 shows the speedup of the `canneal` benchmark on the `Xeon` platform (this benchmark does not compile on the `Power8` platform because the assembler instructions it uses are not available). As for `blackscholes`, the *Map* pattern is spawned as detached. The results obtained are very close to the ones obtained by the PTHREADS and FASTFLOW versions.

4.3 Summary of Results

To summarize the results, we can observe that the PPs proposed in Section 3 and implemented in the CAF framework introduce low overhead and can be profitably used to speed-up performance-critical portion of the application in which the pattern can be used. The shared-memory abstraction, confined within the skeleton implementation of the patterns, gives a significant performance boost to applications if compared with a "pure" AM implementations. This evaluation confirmed that Actors+Parallel Patterns is a flexible parallel programming model capable of obtaining performance comparable to state-of-the-art implementations on multi-core platforms, without renouncing to the features of the AM.

5 Related Work

In the AM, the parallel execution of messages within a single Actor is not allowed. Similarly, data cannot be shared between Actors by construction. These restrictions may lead to scalability issues and to the difficulty of fully exploiting the features of modern shared-memory platforms.

Two distinct approaches are aiming at overcoming these limitations. The first one tries to improve the performance of all mechanisms used to execute Actors efficiently, mainly the Actor scheduling strategies [6, 20, 34, 35]. The second approach, instead, follows the direction of extending the AM with new

features and constructs [10,19,22,25,27,33]. Our work falls in the second category. We added a new parallel abstraction level on top of the AM, providing the user with a set of PPs suitable to efficiently solve a large class of problems and having their implementation skeletons optimized for multi-core platforms.

Concerning the first approach, aiming at optimizing the RTS of Actor-based library, Francesquini et al. [20] designed a NUMA-aware run-time environment based on the Erlang virtual machine. They introduced the concept of *hub Actors*, i.e. Actors with a longer lifespan that create and communicate with many short-lived Actors composing the application. In the proposed system, short-lived Actors are carefully placed on the same NUMA node of hub Actors, thus obtaining an average increase in the application performance. Trider et al. [35] performed a systematic study of the scalability limits of the Erlang language and its VM, presenting a coherent set of technologies, developed within the EU FP7 RELEASE project, to improve its scalability and reliability. Barghi and Karsten [6], proposed an improved version of the Work-Stealing scheduler for the CAF framework, which takes into account locality and NUMA awareness for Actors. The new scheduler offers comparable or better performance than the default CAF scheduler. Torquati et al. [34], the CAF RTS has been optimized to improve the reactivity of Actors and to reduce the latency of messages in streaming applications composed by one or more pipelines.

Concerning the second approach, Scholliers et al. [33] observed that the AM is too strict, and they proposed PAM (Parallel Actors Monitor), a scheduler that expresses a coordination strategy for the parallel execution of messages within a single Actor. Since messages can be processed in parallel within the same Actor in PAM, the programmer is no longer forced to partition the input data to exploit parallelism. The authors of PAM proposed an AmbientTalk implementation of the scheduler, which uses a thread pool inside an Actor. Our approach promotes a more structured approach to these kinds of parallel problems, which on the one hand reduces the programmer's freedom, but on the other hand, provides suitable abstractions with a precise parallel semantics that can be customized and combined to solve the problem at hand.

Some authors proposed to use transactional memory to manage concurrent modifications of the Actor state [22], rather than an ad-hoc internal scheduler [33]. Incoming messages are executed in parallel on a thread pool, and the state modifications are managed as transactions, thus if two modifications conflict, the state is reverted and the modification executed again in a different order. To improve performance in situations where there are many concurrent state modifications, the authors proposed to divide the mailbox into two queues, one for read-only messages (i.e., messages that do not modify the Actor's internal state), and one for the other messages. The two queues are then scheduled on two configurable thread pools to decrease conflicts in the transactional memory.

De Koster et al. [27] proposed a global shared state (the *Domain* abstraction) that the Actors can access to reduce message and synchronization overheads. The authors proposed four different variants of the *Domain*, namely

Immutable Domain, *Isolated Domain*, *Observable Domain*, and *Shared Domain*. Each one enforces different properties and guarantees on the data they manage with respect to the accesses executed by different Actors. Our approach is based on PPs abstractions as the way to encapsulate shared states and to coordinate concurrent accesses according to the parallel semantics of the patterns.

Skel [10] is a parallel library written in Erlang. It provides the user with a set of PPs (e.g., pipeline, farm and seq) 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 main aim of the authors of Skel is to provide a skeleton-based library in Erlang to improve programmability and increase Erlang program performance. Our approach differs because we leverage PPs to address the restrictions of the AM by encapsulating low-level optimizations within the patterns.

In our previous work [32], we proposed to extend the AM with a software accelerator to decrease the execution time of data-parallel computations. We proposed an effective way to partitioning CPU resources between the Actor's scheduler and the data-parallel accelerator. We provided a preliminary implementation of the *Map* pattern that could be executed on the software accelerator. A similar approach was presented in [25], where the authors extend the C++ Actor Framework (CAF) to support external HW accelerator (e.g., GPUs) through OpenCL for speeding up data-parallel computations. 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.

Finally, for the sake of completeness, several works have been made in the context of Active Objects to overcome some of the limitations of the AM. An Active Object (AO) is a pattern of concurrency largely inspired by the AM. An AO runs in its thread of control. The goal is to decouple the object method invocation from its execution to simplify object access [28]. Henrio et al. [23], proposed the Multi-Active Object model, which extends the Active Object model, allowing each activity to be multi-threaded. Hains et al. [21], proposed a new programming model that uses Active Objects to coordinate BSP (Bulk Synchronous Parallel) computations. Fernandez-Reyes et al. [19] proposed an extension of the AO model with the ParT abstraction, capable of running efficient data-parallel computations in a non-blocking fashion with the ability to execute multiple dependent ParT in parallel and to stop the execution on those values that are discovered to be irrelevant for the final result.

6 Summary and Future Directions

In this paper, we discussed the performance limitation of using the "pure" AM in the context of high computational demanding applications on multi-cores. As reported in many research works, and as confirmed by our experimental results, the Actor isolation property may substantially impair the overall performance of Actor-based applications on multi-cores.To overcome this prob-

lem, we proposed to enrich the AM with a set of well-known Parallel Patterns (PPs) capturing recurrent parallel problems. Such patterns are provided to the programmer as a library of highly-optimized skeletons implemented in CAF. PPs transparently encapsulate the use of the shared-memory and all low-level optimizations needed to maximize their performance. At the same time, they offer a clean Actor-like interface that perfectly integrates with the AM without breaking Actor isolation. The resulting programming model combines the best of the two worlds, offering at the user a versatile, safe, and efficient programming environment, also on multi-core platforms.

As future work, we want to extend the PP library with more patterns and test the model on heterogeneous distributed systems.

References

1. Agha, G.A.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
2. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming **7**(1), 1–72 (1997)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: S. Pllana, F. Xhafa (eds.) Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing, chap. 13. Wiley (2017). DOI 10.1002/9781119332015.ch13
4. Allen, J.: Effective Akka: Patterns and Best Practices. " O'Reilly Media, Inc." (2013)
5. Armstrong, J.: The development of Erlang. SIGPLAN Not. **32**(8), 196–203 (1997). DOI 10.1145/258949.258967
6. Barghi, S., Karsten, M.: Work-stealing, locality-aware actor scheduling. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 484–494 (2018). DOI 10.1109/IPDPS.2018.00058
7. Bernstein, P., Bykov, S., Geller, A., Kliot, G., Thelin, J.: Orleans: Distributed virtual actors for programmability and scalability. Microsoft Research (2014). Available: `https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/`
8. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: Characterization and architectural implications. In: 17th Inter. Conf. on Parallel Architectures and Compilation Techniques, PACT '08, pp. 72–81. ACM (2008). DOI 10.1145/1454115.1454128
9. Black, F., Scholes, M.: The pricing of options and corporate liabilities. Journal of Political Economy **81**(3), 637–54 (1973)
10. Bozó, I., Fordós, V., Horvath, Z., Tóth, M., Horpácsi, D., Kozsik, T., Köszegi, J., Barwell, A., Brown, C., Hammond, K.: Discovering parallel pattern candidates in Erlang. In: Proceedings of the 13th ACM SIGPLAN Workshop on Erlang, Erlang '14, pp. 13–23. ACM (2014). DOI 10.1145/2633448.2633453
11. Campbell, C., Miller, A.: A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures, 1st edn. Microsoft Press (2011)
12. Charousset, D., Hiesgen, R., Schmidt, T.C.: Revisiting actor programming in c++. Computer Languages, Systems & Structures **45**(Supplement C), 105 – 131 (2016)
13. Charousset, D., Schmidt, T.C., Hiesgen, R., Wählisch, M.: Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In: Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!, pp. 87–96. ACM (2013)
14. Chasapis, D., Casas, M., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J., Valero, M.: PARSECSs: Evaluating the impact of task parallelism in the parsec benchmark suite. ACM Trans. Archit. Code Optim. **12**(4), 41:1–41:22 (2015). DOI 10.1145/2829952

15. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel computing **30**(3), 389–406 (2004)
16. De Koster, J., Van Cutsem, T., De Meuter, W.: 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, pp. 31–40. ACM Press (2016). DOI 10.1145/3001886.3001890
17. De Sensi, D., De Matteis, T., Torquati, M., Mencagli, G., Danelutto, M.: Bringing parallel patterns out of the corner: The p^{3}arsec benchmark suite. ACM Trans. Archit. Code Optim. **14**(4), 33:1–33:26 (2017). DOI 10.1145/3132710
18. Ernstsson, A., Li, L., Kessler, C.: Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. International Journal of Parallel Programming **46**(1), 62–80 (2018). DOI 10.1007/s10766-017-0490-5
19. Fernandez-Reyes, K., Clarke, D., McCain, D.S.: Part: An asynchronous parallel abstraction for speculative pipeline computations. In: Coordination Models and Languages, pp. 101–120. Springer International Publishing (2016)
20. Francesquini, E., Goldman, A., Méhaut, J.F.: 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, pp. 109–114. ACM (2013). DOI 10.1145/2541329.2541342
21. Hains, G., Henrio, L., Leca, P., Suijlen, W.: Active objects for coordinating bsp computations (short paper). In: G. Di Marzo Serugendo, M. Loreti (eds.) Coordination Models and Languages, pp. 220–230. Springer International Publishing (2018)
22. Hayduk, Y., Sobe, A., Felber, P.: Dynamic message processing and transactional memory in the actor model. In: IFIP International Conference on Distributed Applications and Interoperable Systems, pp. 94–107. Springer (2015)
23. Henrio, L., Huet, F., István, Z.: Multi-threaded active objects. In: R. De Nicola, C. Julien (eds.) Coordination Models and Languages, pp. 90–104. Springer Berlin Heidelberg (2013)
24. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proc. of the 3rd Int. Joint Conference on Artificial Intelligence, IJCAI'73, pp. 235–245. Morgan Kaufmann Publishers Inc. (1973)
25. Hiesgen, R., Charousset, D., Schmidt, T.C.: OpenCL Actors – Adding Data Parallelism to Actor-Based Programming with CAF, pp. 59–93. Springer International Publishing (2018). DOI 10.1007/978-3-030-00302-9_3
26. Imam, S.M., Sarkar, V.: Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In: Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control - AGERE! '14, pp. 67–80. ACM Press (2014). DOI 10.1145/2687357.2687368
27. Koster, J.D., Marr, S., Cutsem, T.V., D'Hondt, T.: Domains: Sharing state in the communicating event-loop actor model. Computer Languages, Systems & Structures **45**, 132 – 160 (2016). DOI https://doi.org/10.1016/j.cl.2016.01.003
28. Lavender, R.G., Schmidt, D.C.: Pattern languages of program design 2. chap. Active Object: An Object Behavioral Pattern for Concurrent Programming, pp. 483–499. Addison-Wesley Longman Publishing Co., Inc. (1996)
29. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Ferret: A toolkit for content-based similarity search of feature-rich data. SIGOPS Oper. Syst. Rev. **40**(4), 317–330 (2006)
30. Mattson, T., Sanders, B., Massingill, B.: Patterns for parallel programming, first edn. Addison-Wesley Professional (2004)
31. Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. " O'Reilly Media, Inc." (2007)
32. Rinaldi, L., Torquati, M., Mencagli, G., Danelutto, M., Menga, T.: Accelerating Actor-based applications with parallel patterns. In: 27th Euromicro PDP Conference, Pavia, Italy, 2019, pp. 140–147 (2019). DOI 10.1109/EMPDP.2019.8671602
33. Scholliers, C., Tanter, É., De Meuter, W.: Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. Science of Computer Programming **80**, 52–64 (2014). DOI 10.1016/j.scico.2013.03.011

34. Torquati, M., Menga, T., De Matteis, T., De Sensi, D., Mencagli, G.: Reducing message latency and CPU utilization in the CAF actor framework. In: 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018, pp. 145–153 (2018). DOI 10.1109/PDP2018.2018.00028
35. Trinder, P., Chechina, N., Papaspyrou, N., Sagonas, K., Thompson, S., Adams, S., Aronis, S., Baker, R., Bihari, E., Boudeville, O., Cesarini, F., Stefano, M.D., Eriksson, S., fördős, V., Ghaffari, A., Giantsios, A., Green, R., Hoch, C., Klaftenegger, D., Li, H., Lundin, K., Mackenzie, K., Roukounaki, K., Tsiouris, Y., Winblad, K.: Scaling reliably: Improving the scalability of the Erlang distributed actor platform. ACM Trans. Program. Lang. Syst. **39**(4), 17:1–17:46 (2017). DOI 10.1145/3107937