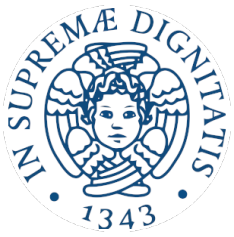
EXTENDING THE ACTOR MODEL WITH PARALLEL PATTERNS: A NEW MODEL FOR MULTI-/MANY-CORE PLATFORMS

LUCA RINALDI



Ph.D. Thesis
Computer Science
University of Pisa

April 2021

Luca Rinaldi, *Extending the Actor Model with Parallel Patterns: a new model for multi-/many-core platforms,* Ph.D. Thesis, © April 2021

SUPERVISORS:
Massimo Torquati
Marco Danelutto

LOCATION:
Pisa

TIME FRAME:
April 2021

ABSTRACT

Multi-/many-core systems have produced a game-changing event for the computing industry. Nowadays, parallel processing is mandatory to improve application performances at all levels of the software stack. However, dealing with concurrency and parallelism is challenging and error-prone. Parallel programmers often face deadlock, starvation, and data-races, and when everything works as expected, sub-optimal performance.

Over the years, research groups and industries have contributed to numerous parallel programming models, languages, and frameworks to ease parallel programming and cope with parallelization issues.

The Actor-based programming model is a well-established model with clear and rigorous parallel semantics. It offers high flexibility for building communication topologies composed of concurrent Actors that can also be spawned and connected dynamically. Actor programmers often pay such great flexibility with additional effort in identifying and solve performance issues and sub-optimal communication patterns in their applications. Furthermore, the tight message-passing semantics of Actors limits optimizations on shared-memory platforms.

On the opposite side, the structured parallel programming approach based on Parallel Patterns offers ready-to-use solutions to recurrent programming problems (e.g., master-worker, pipeline, D&C, map-reduce). The implementation skeletons of those Patterns are usually optimized for the given target platform (e. g., shared-cache multi-cores), thus providing a convenient and straightforward solution to well-known parallel problems. However, the price to pay is reduced flexibility and additional constraints during the software development process. The programmer needs to design the parallel application(s) as a composition of Parallel Patterns. Still, in some cases, it may happen that provided patterns do not support (neither in composition nor alone) the required parallel forms distinguishing the application(s) at hand.

Actors and Parallel Patterns are thus two contrasting approaches to parallel programming. Actors give programmers complete freedom and great flexibility in designing applications but provide sub-optimal performance on multi/many-cores, particularly in data-parallel computations. Instead, Parallel Patterns are less malleable structures targeting specific problems with optimized implementations for a given platform, requiring additional effort and constraints to the programmers during software design and development.

In this thesis, we aimed at building a synergy between the two parallel programming approaches bringing the best of the two worlds into a new heterogeneous yet coherent model, which combines Actors and Parallel Patterns. On the one hand, the introduction of Parallel Patterns in the Actor Model enables the possibility to introduce performance optimizations and structured composition of Actors. On the other hand, the Actor Model provides flexibility and dynamicity features, both necessary in complex parallel applications development.

We analyzed the limitations of the Actor Model on multi/many-cores and investigated multiple approaches to achieve fruitful cooperation of the two models and maintaining, at the same time, the core features and advantages of the two approaches. Through well-known data-parallel and streaming benchmarks, we experimentally demonstrated that the new proposed approach significantly improves the performance of Actor-based applications on multi/many-core platforms. Also, the merge of the two approaches provides the programmer with enhanced flexibility and programmability than the single models. All of this resulted in a new parallel programming library called CAF-PP targeting multi-/many-core systems.

PUBLICATIONS

The full list of my research work published during the Ph.D. period is as follows:

- Luca Rinaldi, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. "High-Throughput Stream Processing with Actors." In: *Proceedings of the 10th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2020. event-place: Virtual, USA. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–10. ISBN: 978-1-4503-8185-7. DOI: 10.1145/3427760.3428338

- Luca Rinaldi, Massimo Torquati, Daniele De Sensi, Gabriele Mencagli, and Marco Danelutto. "Improving the Performance of Actors on Multi-cores with Parallel Patterns." In: *International Journal of Parallel Programming* (June 4, 2020). ISSN: 1573-7640. DOI: 10.1007/s10766-020-00663-1

- Luca Rinaldi, Massimo Torquati, and Marco Danelutto. "Enforcing Reference Capability in FastFlow with Rust." In: *Parallel Computing: Technology Trends, Proceedings of the International Conference on Parallel Computing, PARCO 2019, Prague, Czech Republic, September 10-13, 2019*. Ed. by Ian T. Foster, Gerhard R. Joubert, Ludek Kucera, Wolfgang E. Nagel, and Frans J. Peters. Vol. 36. Advances in Parallel Computing. IOS Press, 2019, pp. 396–405. DOI: 10.3233/APC200064

- Luca Rinaldi, Massimo Torquati, Daniele De Sensi, Gabriele Mencagli, and Marco Danelutto. "Are Actors Suited for HPC on Multi-Cores?" In: 12th International Symposium on High-Level Parallel Programming and Applications. Peer reviewed with internal proceedings. Linköping, Sweden, June 2019, p. 21

- Luca Rinaldi, Massimo Torquati, Gabriele Mencagli, Marco Danelutto, and Tullio Menga. "Accelerating Actor-Based Applications with Parallel Patterns." In: *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). Pavia, Italy: IEEE, Feb. 2019, pp. 140–147. ISBN: 978-1-72811-644-0. DOI: 10.1109/EMPDP.2019.8671602

- Antonio Brogi, Stefano Forti, Ahmad Ibrahim, and Luca Rinaldi. "Bonsai in the Fog: An active learning lab with Fog computing."

In: *Third International Conference on Fog and Mobile Edge Computing, FMEC 2018, Barcelona, Spain, April 23-26, 2018*. IEEE, 2018, pp. 79–86. DOI: 10.1109/FMEC.2018.8364048

- Antonio Brogi, Davide Neri, Luca Rinaldi, and Jacopo Soldani. "Orchestrating incomplete TOSCA applications with Docker." In: *Sci. Comput. Program.* 166 (2018), pp. 194–213. DOI: 10.1016/j.scico.2018.07.005

- Antonio Brogi, Luca Rinaldi, and Jacopo Soldani. "TosKer: A synergy between TOSCA and Docker for orchestrating multicomponent applications." In: *Softw. Pract. Exp.* 48.11 (2018), pp. 2061–2079. DOI: 10.1002/spe.2625

- Antonio Brogi, Andrea Canciani, Davide Neri, Luca Rinaldi, and Jacopo Soldani. "Towards a Reference Dataset of Microservice-Based Applications." In: *Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers*. Ed. by Antonio Cerone and Marco Roveri. Vol. 10729. Lecture Notes in Computer Science. Springer, 2017, pp. 219–229. DOI: 10.1007/978-3-319-74781-1_16

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AM Actor Model

AO Active Object

API Application Programming Interface

BSP Bulk-Synchronous Parallel

CAF C++ Actor Framework

CSP Communicating Sequential Processes

CUDA Compute United Device Architecture

DAG Directed Acyclic Graph

DSL Domain-Specific Language

FF FastFlow

FIFO First In, First Out

GPU Graphics Processing Unit

HPC High-Performance Computing

HW Hardware

LIFO Last In, First Out

MIMD Multiple Instruction stream and Multiple Data stream

MISD Multiple Instruction stream and Single Data stream

MPMC Multi Producer Multi Consumer

MPSC Multi Producer Single Consumer

PCIE Peripheral Component Interconnect Express

PP Parallel Patterns

SIMD Single Instruction stream Multiple Data stream

SISD Single Instruction stream and Single Data stream

SW Software

SPE Stream Processing Engines

SPSC Single Producer Single Consumer

1 INTRODUCTION

In a scenario with pervasive multi/many-core architectures, application performance is not anymore delegated to the single-core computational power. Rather, parallelism is mandatory to conveniently and efficiently exploit the available computing resources and consequently improve application performances at all levels of the software stack [195]. However, dealing with concurrency and parallelism is challenging and error-prone. An application must be analyzed to find tasks that could be executed in parallel, then those tasks need to be distributed on different computational units, and the results need to be collected and merged to produce the final results. All these phases can introduce subtle errors difficult to find, even for parallel programming experts. Furthermore, parallel programmers often face deadlock, starvation, and data-race issues, and when everything works as expected, sub-optimal performance [218].

Over the years, research groups and industries have contributed to numerous parallel programming tools, languages, and frameworks to ease parallel programming and cope with parallelization issues. Some of them rely on sequential program annotations (e. g., OpenMP [68]), others are based on language extensions (e. g., Cilk [37]) or completely new programming languages (e. g., X10 [91]), some others provide libraries of parallel components (e. g., FastFlow [16]) as well as concurrent data structures (e. g., IntelTBB [172]). In the context HPC clusters, the current mainstream programming paradigm is the so-called "MPI + X" approach [202], where the "X" part is a programming model focusing on the single cluster node of the MPI network, for example by using OpenMP or CUDA [183].

In this complex and heterogeneous scenario, a rough but yet significant distinction of programming models is related to how they provide parallelism abstractions to the programmer. In this respect, two opposite classes of models are the *unstructured*, and *structured* approaches to parallel programming. *Unstructured* parallel programming models provide the programmers with primitive concurrent abstractions (such as Threads, Actors, Tasks), which can be used, basically with no restrictions and even dynamically, to build parallel applications. Instead, the *structured* parallel programming approaches leverages more complex atomic structures that can be used to build a communicating topology of concurrent entities (which does not change during the computation). According to this distinction, the Actor Model [7, 122] is a paradigmatic example of the first category.

In contrast, the programming approach based on well-known computational structures called Parallel Design Patterns (or only Parallel Patterns) [148, 149] belongs to the second category.

Both approaches have pros and cons. As we will see, Actors and Parallel Patterns are two examples of models that simplify programmability using completely different approaches.

The Actor-based programming model is a well-established model with clear and rigorous parallel semantics. It was proposed in 1973 by Hewitt, Bishop, and Steiger [122]. The Actor Model offers high flexibility for building communication topologies composed of concurrent Actors. Actor-based applications are characterized by *unstructured* and not rigid communication graphs where Actors can be created dynamically even for a very short life-span. Each Actor can spawn other Actors and can communicate with them just by using their references. Actor programmers often pay such great flexibility with additional effort in identifying and solve performance issues and sub-optimal communication patterns in their applications. Furthermore, the tight message-passing semantics of Actors limits useful performance optimizations for data-parallel computations on shared-memory platforms. However, leaving performance limitations on multi-core platforms aside for a moment [57], the Actor Model is gaining growing success in non-performance critical scenarios thanks to its simplicity and its memory safety guarantees. Indeed, Actors concurrent semantics with its strong memory safety properties also enables its adoption by non-expert parallel programmers. In this respect, the Actor Model has definite advantages [216].

The structured parallel programming approach based on Parallel Patterns [60, 148, 149, 162] offers instead ready-to-use solutions to recurrent programming problems (e. g., *master-worker*, *Pipeline*, *Divide&Conquer*, *Map-Reduce*) [85]. The implementation skeletons of those Patterns are usually optimized for the given target platform (e. g., shared-cache multi-cores as in FastFlow [16]), thus providing a convenient and straightforward solution to well-known parallel problems. However, the price to pay is reduced flexibility and additional constraints during the software development process. The programmer needs to design the parallel application as a composition of Parallel Patterns. In some cases, it may happen that available patterns do not support the required parallel forms needed to solve the problem. Several Pattern-based parallel libraries provide some escape strategies to solve flexibility issues. For example, the FastFlow library has recently provided *building-blocks* abstractions [210] as simple concurrent components that can be assembled to implement custom Parallel Patterns to better fit particular use-cases.

However, finding a good balance between flexibility, programmability and performance in the context of parallel programming is still a challenge.

In this study, we carried out a thorough analysis of how to get the best of the Actor-based and Pattern-based programming approaches. Our aim is to provide the programmer with a synergic integration of the two models suitable to support the development of efficient applications targeting multi-/many-core platforms. We aim at building a new heterogeneous yet coherent parallel programming approach, which combines Actors and Parallel Patterns.

On the one hand, the introduction of Parallel Patterns in the Actor model enables the possibility to introduce performance optimizations at the implementation skeleton of the patterns. In addition, patterns enable introducing well-known structured compositions of Actors (e. g., parallel pipelines). On the other hand, the Actor Model provides the programmer with memory-safety, flexibility, and dynamicity features, all necessary in complex parallel applications development. In our vision, by mixing programming approaches, the programmers can use both models' features and design their applications using Actors and Parallel Patterns together through a unified messaging interface. The result may be summarized as improved flexibility while preserving programmability, increased memory-safety thanks to Actors, and increased performance thanks to specialized implementation skeletons of Parallel Patterns.

Using high-level programming models limits the user to some extent who is led to follow specific idiomatic approaches that could, in turn, lead to performance and programmability issues. Therefore, the programmers may be tempted to skip the model's guidelines and use some kind of custom hand-made implementation to solve their specific problems. An example can be found within the Savina benchmark suite [126], a well-known collection of Actor-based applications. The recursive matrix multiplication benchmark allows Actors to access the matrices concurrently by sharing their references. The implementation is correct, but it breaks the Actor Model isolation principle. The authors implemented the benchmark in that way to improve the performance of the algorithm on shared-memory platforms leveraging low-level features of the Actor Model implementation they used (i. e., the Akka framework [18] written in Java). However, the solution adopted reduces the portability of the benchmark to other languages and frameworks that require more strict guarantees. Moreover, the uncontrolled mix of the Actor Model semantics with the shared-variable approach can be risky for the programmer and error-prone [36].

Similar problems can be found in pattern-based applications. Some stock market analysis financial applications are designed as complex DAGs of computing operators [84]. These applications (a sample application diagram is sketched in Figure 1), can in principle be implemented just using some composition of Parallel Patterns, specifically

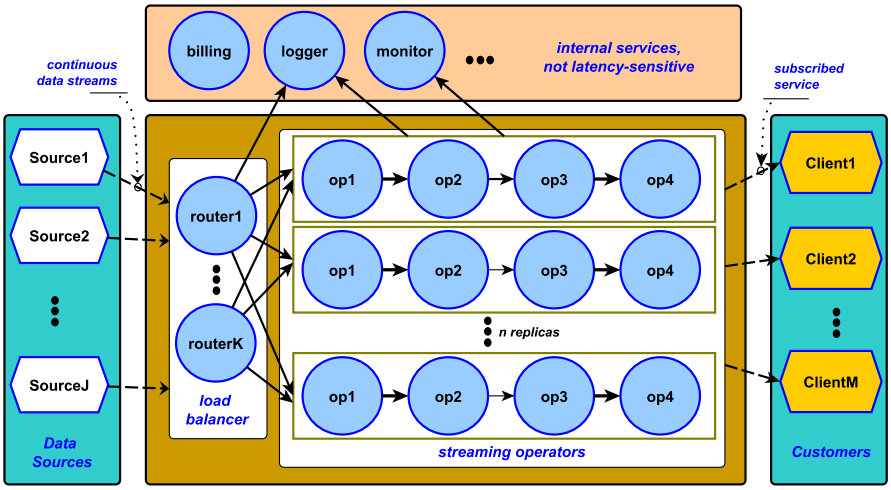


Figure 1: Simplified software architecture of a typical financial application.

parallel pipelines[1] [211]. The input data are processed and aggregated through a network of computing nodes, each implementing a *stateless* or *stateful* operator (e. g., *filter*, *reducer*, *flat-map*). However, the real applications scenario also needs non-performance-critical components that implement specific features that necessitate interaction with other complex systems (e. g., logging, billing, disk data recording). These components are not part of the core application structure but have to be accessible to the application's operators. Current Parallel Patterns frameworks could hardly provide the required flexibility to implement an application with these features, and would need manual intervention to bypass the restrictions imposed by the patterns themselves.

Our idea is to coherently merge the two approaches, that is well-defined Parallel Patterns structures and Actors to improve expressiveness for the programmer without the need for handmade customized implementations.

In the Actors literature, several works aim at improving the Actor Model with the intent of overcoming some of the limitations of the model either for programmability or performance purposes. Most of them introduce some RTS optimizations or add some new features to the model. Instead, others tried to focus on combining the Actor Model with other programming models, e. g., Tasks [197] or the *Bulk-Synchronous Parallel* model [139]. Our work followed a different path, though pursuing similar issues. We aimed at further raising the parallel abstraction level of Actors by leveraging on Parallel Patterns and their optimized implementation skeletons targeting multi-/many-cores.

1 ATS latency benchmark https://github.com/ATS-Advanced-Technology-Solutions/caf_latency_benchmarks

We focus on the viability and the advantages of building a new parallel programming library, which extends the Actor Model semantics with the structured parallel programming methodology based on Parallel Patterns. Although this proposal does not depend on either Actor Model implementations or on the programming languages used, we needed to focus on specific technologies to build a coherent prototype. Our reference Actor Model implementation is the C++ Actor Framework (CAF) [51]. CAF is a high-performance modern C++ implementation of the Actor Model. It follows the *Classic Actor* category, which is closer to Actors' first formalization [83]. Moreover, it provides higher performance than other widely used Actor-based implementations (e. g., Erlang) [50].

The C++ language is nowadays pervasive in high-performance parallel programming. Most of the standard libraries and benchmarks are built using C/C++. Thus, the C++ implementation of CAF eases the comparisons with other specialized libraries targeting multi-/many-cores (e. g., PThreads and FastFlow). Besides, the prototyping work is simplified by using a library, like CAF, build atop a general-purpose programming language such as C++, rather than using Actor Model implementations built atop more specialized programming languages (e. g., Erlang [96] or Pony [165]).

1.1 CONTRIBUTIONS

This thesis's work is centered around defining a new coherent synergy between two parallel programming approaches, namely, Actors and Parallel Patterns, capable of bringing an enhanced trade-off between programmability, expressivity, and performance on modern multi-/many-core platforms.

The thesis summarizes our research work conducted throughout the Ph.D. We organized the thesis according to the research steps we made to achieve the objective.

Figure 2 shows a schema that outlines the main contributions and their interconnections starting from the initial analysis of the Actor Model and the Parallel Patterns programming approach.

All the scientific results obtained have been published in the following list of papers:

- In Rinaldi et al. [175] and in the corresponding extended journal version Rinaldi et al. [176] we propose a set of Parallel Patterns behaving like "macro Actors" and that perfectly integrate with the Actor Model. In the first version, through a set of well-known parallel applications coming from the Parsec benchmark suite, we demonstrated that the Actor Model alone cannot fully exploit the computation capabilities of modern multi-/many-core platforms in data-parallel computations. Instead, by leveraging our Parallel Pattern implementations

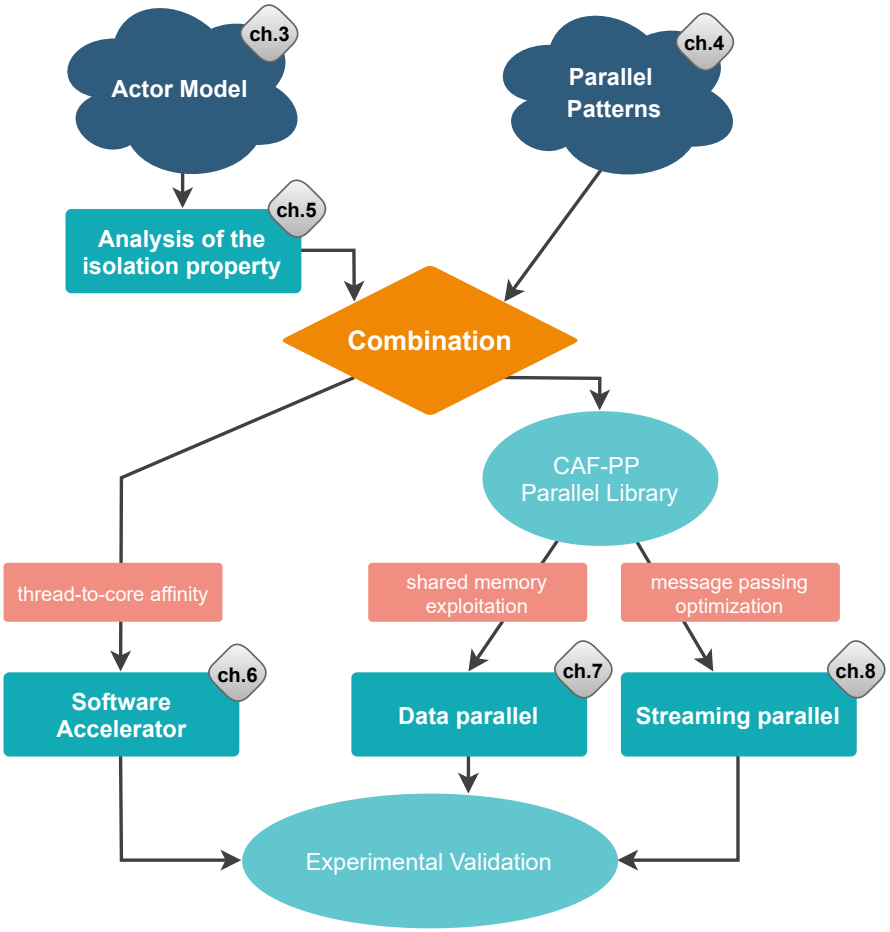


Figure 2: Logical schema of the thesis' content. The rectangles represent the main contributions of the thesis, distinguishing between low-level features in light red and high-level components in light blue. Besides, the gray diamonds connect macro arguments to chapter numbers.

on top of the C++ Actor Framework (CAF) framework, we achieved comparable performance of raw PThreads-based, and FastFlow-based implementations without breaking the Actor Model design principles. Moreover, in the journal version, we also proposed our initial implementation of some *Control-parallel Patterns* to enrich composability and thus programmability in Actor-based applications.

- In Rinaldi et al. [177] we focused instead on high-performance streaming computations. We demonstrated that state-of-the-art C++-based Actor Model implementations like CAF cannot achieve high-throughput and low-latency in higly-demanding streaming computations. To tackle this problem, we extended and enhanced our *Control-parallel Patterns* proposed in Rinaldi et al. [176] to provide optimized implementation skeletons and

their compositions capable of exploiting the resources of modern scale-up servers. We implemented a set of the well-known streaming applications using our Parallel Patterns demonstrating a performance improvement of more than 2×.

- In Rinaldi et al. [178] we introduced our initial attempt to build a software accelerator artifact shared by all Actors of an Actor-based application. The accelerator can speed up the execution of data-parallel computations (e. g., *Map* and *Map+Reduce*). Actors offload their data-parallel computations to the accelerator and wait for the results. We demonstrated that this approach is able to remove the performance problems introduced by bottleneck Actors in data-parallel computations while preserving the flexibility of the Actor Model.
- In Rinaldi, Torquati, and Danelutto [174] we investigated the advantages of introducing statically checked constraints in message-passing semantics. Although this work does not directly target the Actor Model, the analyses we conducted can be read as the base of this thesis work.

The contributions of the thesis can be summarized as follows:

- We prepared a survey of the different Actor Model libraries and languages representing the current state-of-the-art. We classified the languages and libraries according to different implementation approaches and features adopted.
- We investigated the advantage and limitations of the isolation property in the Actor Model. Specifically, we discussed the performance issues introduced by the memory isolation property in data-parallel computations.
- We designed and implemented a shared software accelerator for C++ Actor Framework (CAF) Actors running data-parallel computations.
- We added the thread-to-core affinity feature within the CAF run-time system, enabling the possibility to separate the execution of run-time threads running application Actors in different partitions of the available machine cores.
- We designed a set of Parallel Patterns that integrate with the Actors of CAF. We implemented the open-source CAF-PP C++-based library[2] in which Parallel Patterns are provided to the programmers as "macro Actors".
- We designed and implemented a set of *Data-parallel Patterns* within CAF framework that internally exploits the physical shared-memory in a clean and safe way.

2 https://github.com/ParaGroup/caf-pp

- We designed and implemented a set of *Control-parallel Patterns* within CAF whose composition and nesting can be profitably used to design high-throughput streaming applications with significant performance improvements than the use of sole Actors.
- We ported a sub-set of PARSEC [34] applications in CAF leveraging our CAF-PP library.
- We designed and implemented a straightforward backpressure mechanism for the CAF framework, which simplifies the employment of CAF in high-throughput streaming computations.

1.2 OVERVIEW

To have a quick overview of the thesis's contents, we suggest the reader to consider Figure 2. The figure visualizes how chapters are connected. Indeed, each macro argument in the picture shows a gray diamond with the chapter number in which it is discussed.

The thesis is divided into three main parts. The first part contains the background material and the state-of-the-art. Specifically, the Chapter 2 contains the needed background material to better understand the contributions of the thesis. We define the main characteristics of the Actor Model and of the Parallel Patterns, including information about the library and programming languages that we use in the thesis (e. g., C++ ACTOR FRAMEWORK). Chapter 3 and Chapter 4 propose the state-of-the-art survey related of the Actor Model and the Parallel Patterns programming approach, respectively. We principally analyzed the works that focus on extending the Actor Model or addressing performance limitations of the Actor Model on multi-/many-core platforms.

The second part contains the thesis's main contribution and our proposal to build a new parallel programming approach based on the synergy of Actors and Parallel Patterns. We started analyzing the performance implications of the *isolation* property on multi-/many-cores through different message-passing semantics implementations (i. e., Chapter 5). The *isolation* property is a crucial principle providing the data-race free guarantees of the Actor Model. However, we will also discuss its performance issues on shared-memory systems.

Chapter 6 describes our initial attempt to build the "Actors' accelerator" structure for accelerating data-parallel computations, specifically, *Map* and *Map-Reduce* computations. The proposed accelerator enables speedup in data-parallel computations reducing the service time of bottleneck Actors. In this chapter we also propose the *thread-to-core affinity* feature for the C++ ACTOR FRAMEWORK run-time.

Chapter 7 and Chapter 8 contain our proposal related to integration of Parallel Patterns programming approach into the Actor Model

for multi-/many-core platforms. We propose both *Data-parallel Patterns* and *Control-parallel Patterns* aiming at providing a synergic integration of the two models. Parallel Patterns are "macro Actors" that can be composed and nested together with other application Actors. They cooperate via the standard Actor messaging system. The set of patterns proposed and their optimized compositions can be used to remove performance bottleneck in Actor-based application and implement high-performance streaming networks providing high-throughput and low-latency.

Finally, the third part summarizes the results and outlines possible future research directions (i. e., Chapter 9).

Part I

BACKGROUND AND STATE OF THE ART

2 BACKGROUND

In this chapter, we provide the necessary background that will help understand the motivations and the contributions of the thesis. Section 2.1 presents background information related to parallel programming, starting from parallel architectures focusing on multi-cores, which are the target platforms in this thesis. Then, Section 2.2 discusses the different types of parallelism and Section 2.3 presents some relevant parallel programming models. Finally, in Section 2.4, we present the framework and programming language that we will use in this thesis to do experiments and implement our pattern-based library.

2.1 PARALLEL ARCHITECTURES

A well-known categorization of the Parallel Architectures is based on the number of instruction streams and the number of data streams [102]. In this categorization, the single process system (uniprocessor) has a *Single Instruction stream and Single Data stream* (SISD), and a multiprocessor system has a *Multiple Instruction stream and Multiple Data stream* (MIMD). Indeed, a uniprocessor system has a single stream of instruction and data; instead, in a multiprocessor, each processor fetches its instruction and access its data. Other kinds of architectures are the *Multiple Instruction stream and Single Data stream* (MISD), which are commonly used for fault-tolerance purposes, and the *Single Instruction stream Multiple Data stream* (SIMD), which are the standard architecture of the Graphics Processing Unit (GPU).

Multiprocessors can be further classified in *Symmetric MultiProcessors* (SMP) and *Non-Uniform Memory Access* (NUMA). In an SMP multiprocessor, each process has symmetric access to the memory with the same latency and bandwidth. This is not the case of NUMA multiprocessors. The NUMA architectures also include systems composed of multiple distinct SPM processors connected thought an interconnection network (either a Network-on-Chip or a standard external inter-node network). Multi-cores are typically classified as SPM architectures, even though some multi-core systems are composed of multiple smaller SPM systems packed together and presenting complex memory hierarchies with different access latencies. An example of such system is the AMD EPYC processor [1].

Multi-core architectures start to become popular around 2004 when the Dennard's scaling law [88] stopped working due to severe cur-

rent leakage issues at small sizes and challenging cooling problems raised. Therefore, chip manufacturers moved away from trying to increase the single-core performance over different generations. Dennard's scaling law states that as transistors get smaller, their power density stays constant. This means that chip manufacturers can scale transistors and increase clock speed in each new generation without significantly increasing energy consumption. Dennard's scaling is usually paired with Moore's law [152], which states that the number of integrated circuit resources that can be packed into a single chip doubles approximately every two years. Although the transistor size continues to shrink, at some point, the current leakage poses more significant challenges causing the chip to further heat up. This technology period is commonly known as the "free lunch era" [195], where every few years, the same Software (both applications and system software) can benefit from the performance increase of Hardware upgrade providing faster single processor chip. The jump into the so-called "multi-core era" puts increased pressure on the Software community too. Software performance now mainly depends on the capability of the Software itself to efficiently manage multiple concurrent threads of control running in parallel on different cores to accomplish the same task.

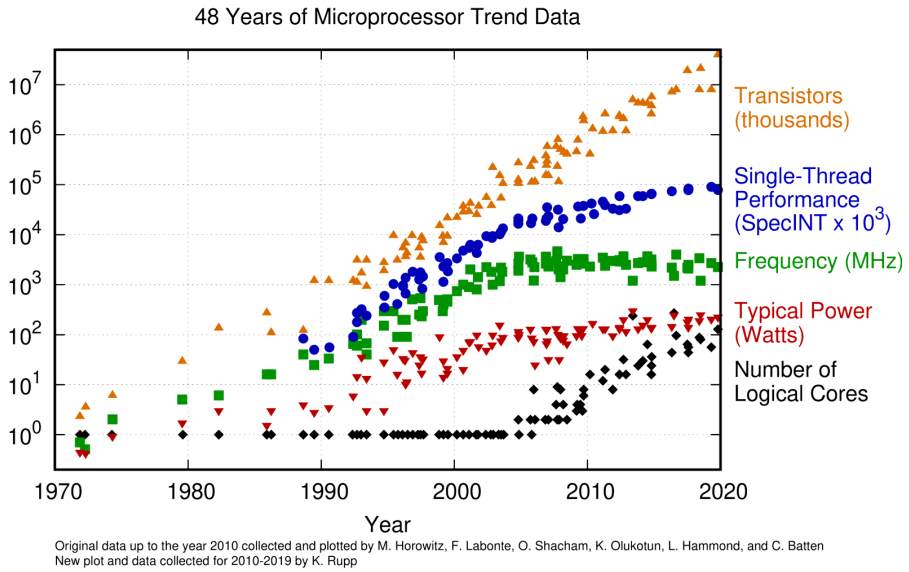


Figure 3: Forty-eight years of microprocessor evolution.
Figure by Karl Rupp [130].

Figure 3 shows the Microprocessor trend of the past 50 years. It is possible to observe that from 2005-2006, the single-thread performance slow down along with frequency and power, while the number of logical cores starts to increase gradually. The number of transistors continues to grow, indicating that Moore's law is still valid. Although, recent works forecast that Moore's law will end around 2025 [203]. To

face this problem, new technologies that will not rely on physical scaling are under investigation (e. g., Spin-based logic [145]).

For the sake of completeness, we also describe other common parallel architectures even though they are not directly considered in this thesis.

GRAPHICS PROCESSING UNIT (GPU)

A GPU is a specialized computer unit initially designed to rapidly manipulate and alter memory to accelerate the image creation in a frame buffer and display it on a display device. GPUs are typically used as coprocessors attached to PCIe buses. Since 2006 they start to be also used as general-purpose computational units (GPGPUs) thanks to the diffusion of more user-friendly programming models such as CUDA and OpenCL. In GPU computing, performance comes from simultaneously using a large number of GPU threads, which are relatively light-weight entities with zero context-switching overhead, to compute the same task on multiple data elements. GPUs are mainly used for data-parallel computations, and the main programming challenge is to reduce (or hide) the time spent transferring data between CPU memory to/from GPU memory. The principal frameworks to run parallel programs on graphic cards are the *Compute United Device Architecture* (CUDA) [183] for NVIDIA GPUs and OpenCL [185], which targets general-purpose hardware accelerators in general (e. g., GPUs, FPGAs). Recently, SYCL [199] proposes a royalty-free and cross-platform abstraction layer for heterogeneous processors in standard ISO C++. It supports a broad variable of target framework, e. g., OpenCL and OpenMP.

DISTRIBUTED SYSTEM

Distributed memory systems (also called multicomputers) are a collection of independent computers, each owning its private memory, and connected through an interconnection network forming a given topology. In the HPC context, the interconnection networks are usually based on high-performance message-passing networks offering both low-latency and high-bandwidth communications such as InfiniBand [164], Myrinet [38] and QSNet [163].

Clusters of Workstations (or simply clusters) are homogeneous distributed systems, in which each cluster node is identical to the others. Internally the nodes are standard multiprocessor servers with their Operating System. On top of that, runs specific cluster software, for example, for accessing the file system (e. g., DFS, NFS), for the execution of jobs (e. g., SLURM [191]), and for providing the programmers with a virtual shared-memory abstraction (e. g., [156]).

2.2 TYPES OF PARALLELISM

It is widely acknowledged that there are two main types of parallelism, *Data Parallelism* and *Task Parallelism*. The first model refers to the exploitation of parallelism by applying the same function on different data elements (e. g., partitions). The second refers to those parallel executions that can be organized on the basis of separate or inter-dependent tasks. *Task Parallelism* can be further distinguished from *Stream Parallelism* if we consider the concepts of a sequence of data elements flowing through a task-dependency graph. A stream is a possibly infinite sequence of values, all of them having the same data type, e. g., a stream of images (not necessarily all having the same format), a stream of files, a stream of network packets, a stream of bits, and so on. In the following, we present the main aspects of the three models:

- *Data Parallelism* is a method for parallelizing a single collection of data by processing independent sub-collections of elements in parallel. The input collection of data is split into multiple sub-collections (or partitions), each one computed in parallel by applying the same function to each partition. The results produced are collected in one single output collection, usually having the same type and size of the input. The computation on the sub-collections may be completely independent, meaning that the computation uses data only coming from the current sub-collection, or it may be dependent on previously computed data. The primary objective of this model is to reduce the completion time (i. e., the total latency) of the entire computation on the initial collection. Particular attention should be put in the workload balance due to the potential variable calculation times associated with each distinct partition.

- *Stream Parallelism* is a method for parallelizing the execution of a stream of elements (or collections of elements) through a series of sequential or parallel modules (sometimes also called operators or filters). Parallelism is achieved by running each module simultaneously on subsequent or independent data elements. In general, this method is applicable if the computation requires a total or partial ordering of different tasks. This method is usually represented as a direct graph in which the edges are the communication channels and the nodes the modules itself.

- *Task Parallelism* is a method for parallelizing the execution of a set of distinct functions by running each of them according to an dependency graph (also referred to as task-dependency graph). In this class, we can also include recursive fork-join computations. Functions (or tasks) are processed concurrently by threads or processes which communicate to satisfy input data

dependency as described by the dependency graph. As for the Data Parallelism method, Task Parallelism is applied to a single data collection.

2.3 PARALLEL PROGRAMMING MODELS

Parallel programming is still frequently approached by using low-level mechanisms and libraries that allow retaining complete control over the underlying platform, e. g., MPI [192], POSIX THREADS [45]. Although this permits to apply platform-specific optimizations (provided that the programmer has enough knowledge of the target platform), it dramatically limits the portability and, in particular, the performance portability. Indeed, the programmer cannot work on his/her algorithms without struggling with threads, synchronizations, and communications concepts, whose features may change over time with novel HW/SW platforms.

To tackle the issues above, high-level programming models provide abstractions that hide away the complexities derived from the direct use of low-level parallelization primitives, thus reducing the time required to implement the application and improve its portability and maintainability. The model's primary aim is to raise the abstraction level, providing the user with useful paradigms and constructs with precise functional and parallel semantics. It follows the *separation of concerns* software design principle, in which system-level programmers build efficient and portable libraries/frameworks, whereas domain expert programmers use these mechanisms to build the application software. Some notable examples of widely used high-level parallel frameworks targeting multi-cores are: INTEL THREADING BUILDING BLOCKS [172], OPENMP [68], and CILK [37].

The programmer can use a composition of those abstractions to build his/her algorithm, confident that the implementation will be compatible with a potentially large variety of architectures. Indeed, the high-level abstractions hide the actual implementations, which use threads and low-level synchronization mechanisms to provide multiple versions with different optimization strategies depending on the underlying architecture. One of the main advantages of using high-level approaches is to reduce programming effort and increase productivity in software development [190]. This approach can also reduce the required skill to exploit parallel programming and thus increasing the proliferation of efficient and highly optimized software [112].

Notable examples of high programming models are the dataflow and the *Bulk-Synchronous Parallel* (BSP). The dataflow model has been formalized by Kahn [129] in 1974. It models a parallel program as a directed graph where operations are represented by nodes while edges model data dependencies. Nodes represent the functional unit

of computations that are fired when all input data items are present. Operations without direct dependencies as well as operations that become fireable at the same time can be executed in parallel.

The BSP model is a general execution model proposed by Leslie Valian in the 90s [214] as an attempt to provide parallel computing with an equivalent to the Von Neumann model for sequential computing. BSP algorithms are defined as a sequence of 3 phases: computation, communication and a global synchronization barrier between all processes composing the application. A group of these phases is called a superstep. In each superstep processes compute using local variable only, then after a global barrier the communication takes place. The messages sent or received during a superstep can be used only after the barrier has been crossed and so at the beginning of the next superstep.

In the following, we discuss two other approaches to high-level parallel programming. The first one is the *Actor Model*, which is a programming model that does not enforce any restrictions to parallel programmers on how to express parallelism. The second one, is the *Structured Parallel Programming Model*, which, instead, limits the freedom to the parallel programmer proposing only a restricted set of parallel paradigms (also called *Parallel Patterns*), with the primary aims of reducing the complexity of the parallelization and providing more opportunities to introduce optimization heuristics.

2.3.1 *Actor Model*

The Actor Model was proposed in 1973 by Hewitt, Bishop, and Steiger [122]. It was initially developed to be used for artificial intelligence research to model systems with thousands of independent processors, each one with local memory and connected through some high-performance network. More recently, the Actor Model gained interest and started to be adopted in the contexts of multi-core processors. The primary principle of the Actor Model is centered on the *Actor* concept. An Actor is an entity that:

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

Indeed, every Actor has a mailbox, and it performs a particular computation considering its internal state, and the message just extracted from its mailbox. The Actor may generate zero, one or more new messages that may be delivered into the mailbox of any Actors whose destination address is known.

The interaction principles among Actors are based on asynchronous, unordered, fully distributed, address-based messaging.

To achieve full asynchronicity, and thus no deadlock, each Actor must have a mailbox of infinite capacity to prevent the generic sender Actor from blocking on a send operation. However, no guarantee is given about the ordering in which an Actor processes the messages present in its mailbox.

Inherently, there is no upper-bound on the time needed by the system to reach some kind of stable state, i. e., a state where all the messages have been processed and no other messages have been sent. The Actor Model only guarantees that all the messages will be eventually processed.

More in detail, the Actor Model is defined by a set of axioms that Hewitt and Barker laid down in their seminal work [121].

One fundamental aspect of the Actor Model is that there is no global state, and no central entity manages the whole system. Actors are memory-isolated entities, and shared mutable states are eschewed in favor of by-value data exchange semantics. 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 new local state of multiple Actors can be computed concurrently. The flow of events in an Actor system forms an activation suborder tree expressing for every event a causality order with a finite path back to the initial event. Different branches of the activation suborder tree represent chains of parallel events.

The Actor Model enforces strict locality by allowing the Actors to build their knowledge about the rest of the system only through messages, including other Actor addresses. 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 other Actor references over incoming messages.

Another essential concept in the Actor Model is the *activity*. An activity is a set of events between the reception of a request and the sending of a replay. Two activities are considered concurrent if their request events occur in parallel, even though the activities may have some overlap. An activity is *determinate* if exactly one replay is generated and *non-determinate* if more than one replay is generated.

Finally, as mentioned by Hewitt and Baker [121] the reliability of the message delivery system is not considered part of the Actor Model. This is considered an orthogonal point, and it implies that in an unreliable network environment, the Actors themselves should take care of the necessary re-transmission and the proper fault handling protocols.

2.3.2 *Pattern-based Parallel Programming*

Parallel programming based on parallel patterns [148] and algorithmic skeletons [60] is a well-recognized approach for raising the level of abstraction in parallel programming. The two approaches define a series of schemes of parallel computations that recur in many applications and algorithms. The programmers can pick, customize, and exploit their well-defined functional and non-functional semantics [14]. Each skeleton could have different implementations depending on the execution architecture, and they also may have different models of execution and coordination.

Pattern-based frameworks provide a set of Parallel Patterns that solve recurrent problems in parallel programming. In the following, we provide an overview of the most used patterns. These parallel patterns encapsulate the algorithmic characteristics of the parallel problem, providing a simple interface focused on reusability and portability. One important strength of Parallel Patterns is the ability to compose one Pattern inside another. Thus, a small set of specific patterns could be composed to solve complex problems in this way.

Each Parallel Pattern implementation library defines different composition rules for their patterns, but one of the most common approaches is the "two-tier model", first introduced in the P^3L language [28, 135], providing two macro-categories of patterns, i. e., *Data-parallel Patterns*, and *Stream-parallel Patterns*. A P^3L valid composition of Parallel Patterns is the one having *Stream-parallel Patterns* at the top of the composition tree and *Data-parallel Patterns* at the bottom of the tree (i. e., they are the leaves of the tree). Thus, the "two-tier model" excludes from valid compositions a *Farm* or a *Pipeline* inside a *Map*, whereas the other way round is valid. This principle is based on the observation that some pattern compositions do not offer any (or minimal) performance improvements while increasing the implementation complexity.

2.3.2.1 *Sequential (`seq`)*

This pattern encapsulates a portion of the business logic code of the application and it is usually used as a parameter of other patterns. The implementation requires to wrap the code in a function $f : \alpha \to \beta$ with input and output parameter types α and β, respectively. For each input $x : \alpha$ the pattern $(\texttt{seq}\ f) : \alpha \to \beta$ applies the function f on the input by producing the corresponding output $y : \beta$ such that $y = f(x)$. The pattern can also be applied when the input is a stream of elements with the same type. Let $stream(\alpha)$ be a sequence $x_1, x_2, ...$ where $x_i : \alpha$ for every i. The pattern $(\texttt{seq}\ f) : stream(\alpha) \to stream(\beta)$ applies the function f to all the items of the input stream, which are computed in their strict sequential order, i. e., x_i before x_j iff $i < j$.

2.3.2.2 *Pipeline* (`pipe`)

The pattern works on an input stream of type $stream(\alpha)$. It models a composition of functions $f = f_n \circ f_{n-1} \circ ... \circ f_1$ where $f_i : \alpha_{i-1} \rightarrow \alpha_i$ for $i = 1, 2, ..., n$. The *Pipeline* pattern is defined as $(\texttt{pipe}\ \Delta_1, ..., \Delta_n) : \texttt{stream}(\alpha_0) \rightarrow \texttt{stream}(\alpha_n)$. Each Δ_i is the i-th stage, that is a pattern instance having input type $\texttt{stream}(\alpha_{i-1})$ and output type $\texttt{stream}(\alpha_i)$. For each input item $x : \alpha_0$ the result out of the last *Pipeline* stage is $y : \alpha_n$ such that $y = f_n(f_{n-1}(...f_1(x)...))$. The parallel semantics is such that stages process in parallel distinct items of the input stream, while the same item is processed in sequence by all the stages.

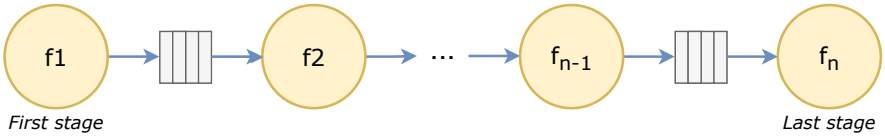


Figure 4: Implementation schema of a *Pipeline* of n stages.

From an implementation viewpoint, a *Pipeline* of sequential stages is implemented by concurrent activities (e. g., threads), which communicate through FIFO queues carrying messages or reference to messages. Figure 4 shows an implementation schema.

2.3.2.3 *Task-Fram* (`farm`)

It computes the function $f : \alpha \rightarrow \beta$ on an input $stream(\alpha)$ where the computations on distinct items are independent. The pattern is defined as $(\texttt{farm}\ \Delta) : stream(\alpha) \rightarrow stream(\beta)$ where Δ is any pattern having input type $stream(\alpha)$ and output type $stream(\beta)$. The semantics is such that all the items $x_i : \alpha$ are processed and their output items $y_i : \beta$ where $y_i = f(x_i)$ computed.

From the parallel semantics viewpoint, within the *Farm* the pattern Δ is replicated $n \geqslant 1$ times (n is a non-functional parameter of the pattern called parallelism degree) and, in general, the input items may be computed in parallel by the different instances of Δ. In the case of a *Farm* of sequential pattern instances, the run-time system can be implemented by a pool of identical concurrent entities (worker threads) that execute the function f on their input items. In some cases, an active entity (usually called *Emitter*), can be designed to assign each input item to a worker, while in other systems the workers directly pop items from a shared data structure.

Output items can be collected and their order eventually restored by a dedicated entity (usually called *Collector*) that produces the stream of results (see Figure 5).

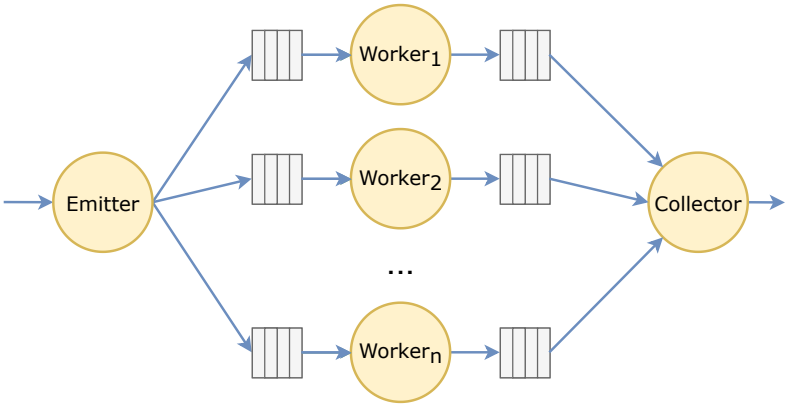


Figure 5: Implementation schema of a *Farm* of n worker.

2.3.2.4 *Master-worker (`master-worker`)*

This pattern works on a collection ($collection(\alpha)$) of type α, i. e., a set of data items $x_1, x_2, ..., x_n$ of the same type $x_i : \alpha$ for any i. There is an intrinsic difference between a stream and a collection. While in a collection all the data items are available to be processed at the same time, in a stream the items are not all immediately available, but they become ready to be processed spaced by a certain and possibly unknown time interval. The pattern is defined as $(\texttt{master-worker}\ \Delta, p) : collection(\alpha) \rightarrow collection(\alpha)$ where Δ is any pattern working on an input type α and producing a result of the same type, while p is a boolean predicate. The semantics is that the `master-worker` terminates when the predicate is false. Different items can be computed in parallel within the `master-worker`.

A master-worker of sequential pattern instances consists of a pool of concurrent workers that perform the computation on the input items delivered by a master entity. The master also receives the items back from the workers and, if the predicate p is true, reschedules some items.

2.3.2.5 *Map (`map`)*

The pattern is defined as $(\texttt{map}\ f) : collection(\alpha) \rightarrow collection(\beta)$ and computes a function $f : \alpha \rightarrow \beta$ over all the items of an input collection whose elements have type α. The output produced is a collection of items of type β where each $y_i : \beta$ is $y_i = f(x_i)$. The precondition is that all the items of the input collection are independent and can be computed in parallel. The run-time of the map pattern is similar to the one described for the *Farm* pattern. The difference lies in the fact that since we work with a collection, the assignment of items to the worker entities can be performed either statically or dynamically. Depending on the framework, an active entity can be designed to assign input items to the workers according to a given policy.

2.3.2.6 *Map+reduction (`map+reduce`)*

It is defined as $(\texttt{map+reduce}\ f, \oplus) : \mathit{collection}(\alpha) \to \beta$, where $f : \alpha \to \beta$ and $\oplus : \beta \times \beta \to \beta$. The semantics is such that the function f is applied on all the items x_i of the input collection (map phase). Then, the final result of the pattern $y : \beta$ is obtained by composing all the items y_i of the output collection result of the map phase by using the operator $\oplus$, i. e., $y = y_1 \oplus y_2 \oplus ... \oplus y_n$.

A typical implementation is the same as the map where the reduction phase can be executed serially, once all the output items have been produced, or in parallel according to a tree topology by exploiting additional properties of the operator $\oplus$ (i. e., if it is associative and commutative).

2.3.2.7 *Divide&Conquer (`D&C`)*

The *Divide&Conquer* algorithms are composed of two distinct phase, the *Divide* (or *Split*) and The Conquer (or Merge) phase. In the former the problem is recursively decomposed into smaller sub-problems building a tree of calls. In the latter 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 *Divide&Conquer* 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. Indeed, 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.

Formally the pattern can be defined as $(\texttt{D\&C}\ d, b, c) : \alpha \to \beta$. The function $d : \alpha \to (\alpha_1 ... \alpha_n)$ divide the problems in sub-problems with the same type of the original problem. The function $b : \alpha \to \beta$ compute the trivial base case, namely the result of a single small enough sub-problem, and finally, the function $c : (\beta_1 ... \beta_n) \to \beta$, recombine the results.

A common optimization of the *D&C* parallel patter consist on early stop the recursive division of the problem in sub-problems, using a configurable *cut-off* value [70, 148]. This permits to execute a sequential version of the algorithm when the sub-problem can fit in cache and it is quite trivial to be computed by a single concurrent entity.

In the Algorithm 1 there is the schema of the sequential `solve` function that the pattern can use to compute the `Solution` type recursively starting from the `Problem` type.

2.4 USED LIBRARY AND PROGRAMMING LANGUAGES

In this section, we propose the technologies that we adopted to build the software prototypes presented in this thesis, namely Rust, and

Algorithm 1: Sequential pseudocode of the *Divide&Conquer* algorithm

```
   Input: Problem
   Output: Solution
1  Function solve(p : Problem) → Solution
2      if isBaseCase(p) then
3          return base(p);
4      else
5          subProblems ← split(p);
6          subSolutions ← ∅;
7          foreach p ∈ subProblems do
8              subSolutions ∪ solve(p);
9          end
10         return merge(subSolutions);
11     end
12 End Function
```

C++ Actor Framework. We use C++ Actor Framework (CAF) as a reference Actor Model implementation. CAF is a high-performance C++ implementation of the Actor Model. The C++ language permits us to both integrate into the framework some low-level optimization and also to fairly compare its performance with reference parallel library not based on the Actor Model, e. g., FastFlow. We uses CAF as base of our prototype in Chapter 5, Chapter 6, Chapter 7, and Chapter 8.

Instead, Rust is a modern system language that implements a strong type system. We adopt Rust to build a safe and efficient message passing mechanism in Chapter 5.

2.4.1 C++ Actor Framework (CAF)

The C++ Actor Framework (CAF) [51, 52, 123] enables the development of concurrent programs based on the Actor Model leveraging modern C++ language. In contrast to other well-known implementations of the Actor Model, such as Erlang [24] and Akka [18], which use virtual machine abstractions, CAF is entirely implemented in C++, and thus CAF applications are compiled directly into native machine code. This allows using the high-level programming model offered by Actors without sacrificing performance introduced by virtualization layers.

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 run-time includes a scheduler that dynamically allocates ready Actors to workers. Whenever a waiting Actor receives a message, it changes its internal state to ready and the scheduler assigns the Actor to one of the worker thread 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 threads workload or starvation. The CAF programmer can explicitly detach Actors by using the detached spawn option, so that the Actor lives in a dedicated thread of execution. A particular kind of detached Actor is the blocking Actor. Detached Actors are not as lightweight as event-based Actors.

In CAF, Actors are created using the `spawn` function. It creates Actors either from functions/lambdas or from classes and returns a network-transparent Actor handle. Communication happens via explicit message passing using the `send` command. Messages are buffered in the mailbox of the receiver Actor in arrival order before they are processed. The response to an input message can be implemented by defining behaviors (through C++ lambdas). Different behaviors are identified by handler function signature, for example using atoms, i. e., non-numerical constants with unambiguous type.

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

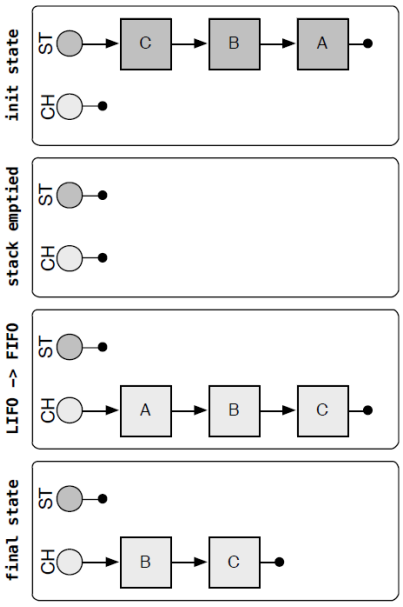


Figure 6: Dequeue operation in CAF queue [52].

tor inserts new elements atomically in the head of the LIFO queue. The receiver Actor, atomically extracts all the messages from the LIFO queue using a *compare-and-swap* operation. Then, the messages are divided into the two FIFO queues on the basis of their priority. Finally, the consumer Actor can dequeue messages with a different proportion from the two queues to maintain the priority.

Figure 6 shows the different phase of the message dequeuing operation considering a situation with a single priority FIFO queue. The initial situation has three elements in the LIFO queue and no elements in the FIFO queue. In the second state the LIFO queue is emptied by a single operation. In the thread state all the elements are inserted backward in the FIFO queue, and finally in the last state the first element was popped.

Listing 1 presents a simple example showing some of the features of the CAF framework, e. g., the dynamic behavior changing. Two Actors exchange an integer value that is decremented until it becomes zero. Line 38 declare the main function that will be call by the CAF framework after the `actor_system` creation and initialization. From the same `MyActor` definition, Line 11, the `actor_a` and `actor_b` are spawned at line Line 40 and Line 41, respectively. The second Actor is spawned as detached Actor while `actor_a` is an event-based Actor. `MyActor` define at first a single behavior for the `init_a` atom, using C++11 lambda at Line 15. Instead, Line 13 configures the Actors to skip and to keep in the queue all messages that do not match the defined behaviors. Thus, in this case all messages except for `init_a` messages. After the Actors receive the init message they change their behaviors, by using the `become` method at Line 22. The new supplied behaviors are `send_a` and `stop_a` defined at Line 23 and 23 respectively. Then, by using the send behavior (Line 27), the two Actors exchange the integer value `n` each time decrementing it until it becomes `0`. Eventually, the stop behavior is fired, which terminates the current Actor only after having sent the other peer the stop message.

2.4.2 *Rust Programming Language*

Rust [133, 146] is a modern system-level programming language that focuses on memory safety and performance. Its design gives access to some low-level features with high-level safety. The Rust project was initially started by Graydon Hoare and in 2009 has been sponsored by Mozilla. Since then, Mozilla started developing Servo, an experimental browser engine entirely written in Rust and then integrating some of the stable features in the Gecko browser engine, implemented in C++. In 2017 the Servo CSS style engine was integrated in Gecko and release in Firefox 54. Things change during the COVID-19 pandemic, when Mozilla Foundation lay off around 250 people, including most of the developer of Rust products [49]. That situation create uncer-

```
#include <caf/all.hpp>

using namespace caf;

// define custom atom types for message pattern matching
using init_a = atom_constant<atom("init")>;
using send_a = atom_constant<atom("send")>;
using stop_a = atom_constant<atom("stop")>;

// actor implementation
behavior MyActor(event_based_actor *self) {
  // skip messages until we receive a init atom
  self->set_default_handler(skip);
  // return the (initial) actor behavior
  return {[=](init_a, actor next, int n) {
    // restore default action to printing unhandled messages
    self->set_default_handler(print_and_drop);
    // start sending
    if (n > 0) {
      self->send(self, send_a::value, n);
    }
    self->become(
       [=](send_a, int n) {
         if (n == 0) {
           self->send(self, stop_a::value);
         } else {
           self->send(next, send_a::value, n - 1);
         }
       },
       [=](stop_a) {
         self->send(next, stop_a::value);
         self->quit();
       });
  }};
}

// declare the main function of the program
void caf_main(actor_system &sys) {
  // spawning actors
  auto actor_a = sys.spawn(MyActor);
  auto actor_b = sys.spawn<detached>(MyActor);

  // initialize actors
  anon_send(actor_a, init_a::value, actor_b, 100);
  anon_send(actor_b, init_a::value, actor_a, 0);
}
CAF_MAIN()
```

Listing 1: A simple CAF example that uses dynamic message handler.

```
1 let s1 = String::from("The answer to the ultimate question of life is");
2 // the ownership is transferred from s1 to s2
3 let s2 = s1;
4 // compiler error, s1 is no more accessible
5 println!("{} 42!", s1);
```

Listing 2: Rust ownership example.

tainty and confusion on the future of Rust programming language. Thus, the Rust Core Team and Mozilla itself announce the creation of an independent Rust Foundation, which give complete autonomy from Mozilla starting from the end of 2020 [138].

The principal novelty of Rust is in the management of memory. Languages like C/C++ provide the user with total control on memory allocation and deallocation. Programmers can create, destroy and manipulate the memory space without any limitation. This is a very attractive feature for expert programmers, but it can also lead to very subtle bugs and vulnerabilities (e. g., buffer overflow). Other popular languages such as Java, rely on a Garbage Collector (GC) to safely manage memory without the explicit intervention of the user. The increased security comes along with some performance degradation due to the GC service running in the background trying to reclaim unused memory. Instead, the Rust language deals with memory management through the *ownership* concept [132]. The compiler statically checks a set of rules to control the memory allocation/deallocation and memory accesses. Therefore, the compiler guarantees a certain level of memory safety at the price of a more complex and longer compilation process but without any additional overheads at running time.

Concerning the *owenship* feature, once a variable is bound with a value, it gains exclusive ownership of it. Therefore, only the owner can access that memory location until it transfers the exclusive ownership to another variable. The *ownership rule* states three simple concepts [132]:

1. each value has a variable that is called *owner*;
2. there can be only one owner at a time;
3. when the owner goes out of scope, the value will be dropped.

In Listing 2 there is a Rust snippet of code that does not compile because in Line 5 the ownership rule is violated. In fact, in Line 3 the ownership of variable `s1` is transferred to variable `s2`, and, therefore, the original owner (`s1`) cannot access it again on Line 5.

Values stored in the heap maintain the same rules and when the owner variable goes out of scope the memory is automatically released. In this way the user does not have to directly deal with allo-

cation and deallocation instructions avoiding the risk of double frees or memory leaks.

To improve the flexibility of the language, Rust also implements the *borrowing* concept through memory references. It is possible to create an immutable reference by using `&` and a mutable reference by using `&mut`. Both of them borrow the value from the original owner. The compiler imposes the following rules:

1. at any given point in time, only one mutable reference or any number of immutable references may exist;
2. the borrowed value cannot be accessed by the original owner;
3. when the reference goes out of scope the ownership goes back to the original owner.

```
1  fn main() {
2   // define a mutable variable vec1
3   let mut vec1 = vec![1, 2, 3, 4];
4   // borrow a mutable reference
5   inc_vector(&mut vec1);
6   // the ownership come back to vec1
7   // borrow an immutable reference
8   let sum = sum_vector(&vec1);
9   // the ownership come back to vec1
10 }
```

```
11 fn inc_vec(v: &mut[i32]) {
12   v.iter_mut()
13     .for_each(x *x += 1);
14 }
15
16 fn sum_vec(v: &[i32]) -> i32
       {
17   v.iter().sum()
18 }
```

Listing 1: Example of the Rust *borrowing* feature.

In Listing 1 there is a demonstration of how mutable and immutable references work. In Line 5 a mutable reference is provided to the function `inc_vector` and on Line 8 an immutable reference is provided to the function `sum_vector`. In both cases, the function borrows the value from the vector variable, and when it finishes the execution, the ownership of the vector came back to the original variable `vec1`.

Rust has also the *lifetimes* concept to avoid dangling references. A lifetime is the scope in which a reference is valid and the compiler enforces that it must be smaller of the scope of the value referenced. Lifetimes are usually inferred by the compiler and could be omitted. However, there are cases in witch the user has to annotate references with life time parameters. Life time parameters are defined by an apostrophe and a letter, e. g., `'a`.

Finally, Rust provides native threads support, synchronization mechanisms such as *mutex* and *atomic* variables as well as *Multi-Producer Single-Consumer* (MPSC) communication channels for connecting threads. Indeed, the compiler guarantees that either multiple threads have only read access to a memory location or only one thread has read and write access to it. To manage the mutability

of variables and to guarantee memory safety in multi-thread applications, Rust defines the `Send` and `Sync` traits. The `Send` marker trait indicates that ownership of the type implementing `Send` can be transferred between threads. The `Sync` trait indicates that it is safe for the type implementing `Sync` to be referenced from multiple threads. Almost every primitive types in Rust implements `Send` and `Sync` and types composed entirely of types that are `Send`-able or `Sync`-able are themselves `Send`-able or `Sync`-able. Major exceptions includes, raw pointers and `Rc` that are neither `Send` nor `Sync`. The `Rc` is a reference counter pointer, similar to a C++ `shared_ptr` but without the atomic reference counting operations, therefore not safe in multi-thread sharing.

3

STATE OF THE ART: ACTOR MODEL

In this chapter we describe some state of the art solutions for the Actor Model, which will help to understand the contributions of this thesis.

The Actor Model has a long history of research activity starting from Hewitt, Bishop, and Steiger's first paper in 1973 [122] and then formalized in Hewitt and Baker [121]. During this period of almost fifty years, the Actor Model has been investigated and adopted both in academia as well as in industry with several implementations. Relevant researches were conducted from both a theoretical and a practical standpoint, studying the model as a powerful concurrent abstraction.

For instance the work of Clebsch et al. [57] used the Actor Model in the context of *type systems* and *capabilities*, the work of de'Liguoro and Padovani [86] used the Actor Model in the field of *behavioral types*. Talcott [200] and Gaspari and Zavattaro [107] investigate process algebra and rewriting logic in the Actor Model respectively. Conversely, more technical works are those of Imam and Sarkar [126] and Blessing et al. [36] that propose some benchmarks to compare the performance of different Actor Model implementations.

In this vast research area, we will discuss particularly related works that focus on extending and improving the Actor Model to enhance its non-functional aspects (i. e., performance and programmability) (Section 3.1). We will also present a fraction of the research activity related to concurrency models, which studied the pros and cons of the Actor Model (Section 3.2). Then, we discuss some related work on the so-called Active Objects model, a new model inspired by the Actor Model (Section 3.3). Finally, we present a landscape of the most relevant and recent programming languages and libraries that support the Actor abstraction, dividing them into five distinct categories (Section 3.4).

3.1 ATTEMPTS TO IMPROVE THE ACTOR MODEL

The Actor Model is considered a powerful concurrent model capable of providing important guarantees to deadlock-freedom and data-race-freedom. However, the implementation of those guarantees may sometimes introduce some performance overheads. In this section, we present some research works that propose extensions/improvements to the Actor Model to mitigate such implementation overheads. Some of them try to provide more efficient Run-time System mechanisms

to speed up the execution of Actor-based applications, others, instead provide high-level abstractions to support the distinctive features of the Actor Model.

We classified the research papers that tried to improve the Actor Model in five distinct categories on the basis of either their work objective or on the performance issues they tried to address:

- works that target mechanisms used to execute Actors efficiently (e. g., the Actor's scheduling strategies),
- works that try mixing the Actor Model with other concurrency models (e. g., Actors and Task-Parallelism),
- works that enable concurrency inside a single Actor entity,
- works that extend the model with missing features, primarily in the direction of safely using shared states among Actors,
- works that target Actor Model benchmarking with the aim of providing a common ground for evaluating Actor Model implementations

The remainder of this section describes the most relevant recent works for each one of the aforementioned categories.

3.1.1 *Improving efficiency in the Actor Model*

The first group of papers aims to optimize the Run-time System of Actor Model languages/libraries without introducing disrupting changes in the Actor Model itself to be entirely transparent to the users.

Francesquini, Goldman, and Méhaut [104], Trinder et al. [213] and Barghi and Karsten [29] proposed an improvement of the run-time scheduler of the Actor Model. Francesquini, Goldman, and Méhaut [104] 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. Trinder et al. [213] performed a systematic study of the scalability limits of the Erlang language and its virtual machine, presenting a coherent set of technologies, developed within the EU FP7 RELEASE project, to improve its scalability and reliability. Barghi and Karsten [29], proposed an improved version of the Work-Stealing scheduler for the C++ Actor Framework, which takes into account locality and NUMA awareness for Actors. The new scheduler offers comparable or better performance than the default C++ Actor Framework scheduler.

The works of Torquati et al. [211] and Bauer and Mäkiö [32] focus on improving the message-passing performance of the Actor Model. Torquati et al. [211] optimize the C++ ACTOR FRAMEWORK Run-time System to improve the reactivity of Actors and to reduce the latency of messages in streaming applications composed by one or more pipelines. The generic Worker thread of the C++ ACTOR FRAMEWORK Run-time System leverages a *condition variable* in combination with short waiting time in the `wait_for` methods, in order to maintain a balance between reactivity and resource usage. Bauer and Mäkiö [32] design ACTOR4J a new Actor Model implemented in JAVA that focuses on optimizing message exchange among Actors. ACTOR4J differently from AKKA moves the Actor queue to the underline native executor, thus an Actor is bound to a specific executor together with other concurrent Actors[1]. Using this approach, the implementation can optimize message-passing at the level of the underline executor avoiding synchronizations costs on Actors associated on the same thread.

Another Run-time System targeting enhanced performance for the Actor Model is the one proposed by Desell and Varela [89]. The authors developed a new run-time for the Actor Model language SALSA, called SALSA LITE. The run-time is designed to implements the basic features of the Actor Model, i. e., message-passing communications and dynamic Actor creation in an efficient way. SALSA LITE is highly optimized for multi-core platforms and proposes a non-transparent execution of Actors in which the user should manually assign Actors to executors. Both SALSA LITE [89] and ACTOR4J [32] introduce specific optimizations in the message-passing implementation by removing transparent Actors scheduling. These works demonstrate that the decoupling of lower-level executors and high-level Actor entities may introduce significant efficiency issues on multi-cores.

Similarly, AL-Twajre [10] assess the mailbox performance and the features of different SCALA Actor Model languages. The authors found that SCALAZ [184], an extension to the core Scala library for functional programming, offers the best message-passing performance. Nevertheless, AKKA offers the most future rich framework for both single machines and distributed systems.

Finally, Clebsch and Drossopoulou [56] proposes a novel approach for the implementation of the garbage collection of dead Actors in a fully concurrent way. The main innovation was the possibility to identify reference cycles of dead Actors in a non-blocking manner through messages. Aumayr et al. [25] introduced the ability to create an asynchronous and distributed snapshots in an Actor application. The approach does not use heavy-wait synchronizations, and therefore it does not introduce a significant latency increase in the execution. The snapshots feature is provided for SOMMs, an implementation of the

1 This is similar to the *Communicating Event-Loop* approach in managing Actor concurrency, see Section 3.4

Newspeak Actor Model language based on the *Communicating Event-Loops* paradigm.

3.1.2 *Mixing Actors with other concurrency models*

The second group of works mixes the Actor Model with other programming models to either provide new abstractions to the Actor Model or to adapt it to a different hardware/software platform.

Imam and Sarkar [125], proposed a controlled way to use Task-Parallelism together with the Actor Model. The authors focus on the limitation of the Actor Model, raising the problem of synchronizing multiple concurrent Actor activities explicitly. This is the case, for example, when an Actor needs to distribute some works to other Actors and then waits for their completion (i. e., data-parallel computations). The authors addressed this problem by enabling Task-Parallelism to be executed along with Actors on the same Run-time System. Therefore, developers could mix Actors with the proposed `async-finish` primitive that can create and wait for Tasks. They also proposed to spawn multiple parallel Tasks during Actor message handling with the constrain that the Actor is suspended until all spawned Tasks finish. As we shall see in the next section, relaxing the Actor Model constrain of sequentially executing a single message could complicate the Actor state managing and potentially produce race conditions.

Swalens, De Koster, and De Meuter [197] extends the works of the previous paper [125] integrating *Transactional Memory*, Actors and Futures[2]. The authors propose the Chocola framework based on a fork of the Clojure programming language, which combines Futures, Actors, and *Transactional Memory*. Chocola tries to maintain the semantics and guarantees of the three models and guide the users to correct compositions avoiding the risk of introducing errors.

Futures *are placeholder variables that represent the result of concurrent computations, e. g., a Task. In the beginning, the* Futures *is unresolved. Then, when the Task yields a value, the* Futures *is elected to that value.*

Transactional Memory *is a concurrency control mechanism analogous to database transactions, which allow a group of load and store instructions to execute in an atomic way.*

The CAL Actor Language [93] is an attempt to integrate the Dataflow programming model [128] with the Actor Model. CAL has been adopted in the standardization effort of the MPEG Reconfigurable Video Coding Framework [33]. The new proposed model is based on Dataflow Actors, which are lightweight static entities with a set of input and output communication ports. Those Actors can be statically composed in an acyclic graph and then executed on some native executors. CAL drops some key features of Actor Model, e. g., dynamic spawn of Actors, and dynamic message communication. Conversely, it relies on static connections among Actors that will be feed with streams of data.

Similar works are the stream-oriented extensions for Akka and C++ Actor Framework. Akka Stream [77] is an extension of Akka targeting streaming applications. The library is based on the *Reactive*

2 In this case, we consider Futures and Task-Parallelisms closely related, but they could also be used independently.

Stream project and implements a static graph of streaming operators working on typed messages.CAF STREAM [194] experimental extension of the C++ ACTOR FRAMEWORK has a similar design to AKKA STREAM, but (for now), it does not provide any high-level structure for building networks of streaming operators. Differently from AKKA STREAM, it maintains a close connection with standard C++ ACTOR FRAMEWORK Actors.

A different approach is that of Hiesgen, Charousset, and Schmidt [123], which extends the C++ ACTOR FRAMEWORK library to support external hardware accelerators (e. g., GPUs) through OpenCL [157]. This approach permits executing data-parallel code on hardware accelerators within the scope of an Actor to speed up its execution. The extension implements an OpenCL manager and a new OpenCL Actor. The manager supports the interaction with OpenCL capable devices and it can spawn OpenCL Actors. Although this approach can only enable data-parallel computations leveraging additional hardware, it is beneficial given the increasing number of OpenCL capable devices (e. g., GPUs, FPGAs). However, device management is left to the users without additional abstraction.

An high-level abstraction on top of the Actor Model is the one proposed by the SKEL [39] parallel library for ERLANG. The library provides the user with a set of high-level Algorithmic Skeletons [59] (i. e., pipeline, and task-farm) that can be composed in a functional way. 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 for some specific parallel patterns.

While in the previously mentioned works, the authors brought a different programming model beside or atop of the Actor Model, in the work of Shali [187], Roloff et al. [180], and Pöppl, Baden, and Bader [166] the authors followed an opposite approach. They implemented the Actor abstraction concept in the *Partitioned Global Address Space* (PGAS) programming model. Shali [187] designed an Actor abstraction for the CHAPEL language. Roloff et al. [180] designed and implemented the *actorX10* library for the X10 language. Finally, Pöppl, Baden, and Bader [166] create an Actor library on top of the PGAS library UPC++ implement in modern C++. All papers concluded that integrating the Actor Model in the PGAS model can bring multiple advantages in terms of programmability and performance [167].

In addition, Crafa and Tronchin [64] compares the PGAS model with the Actor Model. The authors wanted to assess the most convenient model in the context of Big Data Analytics between the two models. For doing that, the authors implemented from scratch, by using X10 and AKKA, two of the most used paradigms in Big Data Analytics, i. e., *MapReduce* and *Bulk Synchronous Parallel* (BSP). Then they evaluate both performance and programmability. Their principal findings were that the centralized and imperative flavor of X10 stands

out in the MapReduce implementation, while Akka's Actors foster the distributed execution of asynchronous Tasks better, enabling the scaling to higher concurrency degree as required by BSP.

3.1.3 *Concurrency inside a single Actor*

The third group of research articles contains those works that tried to relax the constraints of having the Actor as an indivisible concurrent entity that fetches messages one by one from its input mailbox and processes it sequentially.

The Actor Model avoids data race and maintain actor isolation by processing one message at a time. Scholliers, Tanter, and De Meuter [186] proposed a customizable message scheduler that is capable of scheduling the processing of multiple not-depending messages at the same time. The typical example is an Actor cell that may receive read and write requests. The Actor cell can process multiple read operations concurrently, while write operations require exclusive execution by the Actor. The authors also proposed an AmbientTalk [65] implementation, which uses a Thread Pool inside the Actor to implement the concurrency.

Hayduk, Sobe, and Felber [117] provided a different approach to enable concurrency inside each Actor based on *Transactional Memory* [116]. Incoming messages are executed concurrently by a Thread Pool, and the state modifications are managed as transactions, thus if two write operations conflict, the state is reverted and the operations are executed again in a different order. To improve performance in the situation when there is a lot of concurrent state modifications, the authors also proposed to split the input mailbox into two queues, one for read-only messages (i. e., messages that do not modify the Actor internal state), and the other for all other kinds of messages. The two queues are then scheduled on two configurable Thread Pools to decrease conflict in the *Transactional Memory* and increase the overall performance.

Finally, Azadbakht, Boer, and Serbanescu [26] designed a new programming concept called Multi-threaded Actor (MAC), which features Actors as a group of Active Objects sharing a message queue. When an Active Objects fetches a message from the shared message queue, the object executes the corresponding thread in parallel with all other threads. Furthermore, MAC supports a mechanism of synchronized data to constraint the parallel execution of messages. Each message provides a set of locked data requests that enable or deny specific messages' concurrent execution. The authors provided a formal operational semantics of MAC and a simple Java implementation of the central programming abstraction.

To summarize, the different research proposals presented in this section [26, 117, 186], try to parallelize heavy Actor computations

without splitting them into multiple small Actors. This permits avoiding to split the Actor internal state and to induce complex message-based protocol implementations for the internal state updates. However, in some situations, this approach could increase application performance but require to rely on suitable abstractions for state management to not impair programmability. Nevertheless, implementing this parallelization using native Threads could definitively reduce the flexibility and dynamicity of the Actor Model.

3.1.4 *Improve Actor Model with new features*

The fourth group of papers focuses on extending the Actor Model to make it more powerful and usable.

There are a set of attempts that try to provide Actors with the ability to share data safely and reliably. De Koster, Marr, and D'Hondt [80] introduce *synchronization views* as a way of synchronizing access to a remote resource owned by another Actor. Views are objects shared between multiple Actors that maintain consistency through synchronized data access. The authors propose two kinds of views: a shared view and an exclusive view, which mimic the well-known multiple reader/single writer synchronization protocol. The views implementation focuses on maintaining the same most important guarantees of the Actor Model: race-condition freedom, and deadlock freedom. Later, the same authors extends the View idea with the *Domains* [81, 82]. This mechanism is a more complete and elaborated solution that supports different kinds of shared states, namely *immutable domain*, *isolated domain*, *observable domain*, and *shared domain*. A domain is a heap object that can be accessed by multiple Actors thought an asynchronous request, which returns a *synchronization view* of the objects. After a request, the data object can be accessed synchronously by the defined policy. Finally, the same authors also proposed another approach that avoids the first asynchronous call to access the reference to the shared state [79], i. e., *Tanks*. A *Tank* is an Actor that can share to other *Tanks* part of its internal state in read-only mode. The proposed solution uses *Software Transactional Memory* [120] to enable readers' operations to execute in parallel with writers' operations, thus avoiding to block the readers while all asynchronous communications were removed.

Prokopec and Odersky [168] and Imam and Sarkar [127] define two new models that extends the Actor Model. In particular, Prokopec and Odersky [168] propose the Reactive isolated model, where the concept of Actors is extended with the concept of *Reactors*. Reactors are entities that listen to multiple kinds of events (e. g., a combination of multiple message reads) and react to them taking actions. The model supplies code reuse and composition, which are well-known

issues of the Actor Model, allowing the composing of the Reactor with multiple distinguished and composable protocols.

Imam and Sarkar [127] present the *Selectors* another extension of the Actor Model. The *Selectors* manages multiple incoming mailbox that can be activated or deactivated by the Selector. A deactivated mailbox may continue to receive messages, but the *Selectors* will temporarily ignore it. The authors claimed that the use of the SELECTORS concept makes synchronization and coordination patterns more natural to implement (e. g., synchronous request-reply, producer-consumer with bounded buffer).

Finally, Gruber and Boyer [114] proposed a run-time ownership-based system to endure Actor isolation without forced a deep-copy of all messages. Actor isolation is implemented by keeping track of the message owner and invalidate references to the message of other Actors to inhibit its access. The author proposes a prototypes developed on top of KILIM [131] Actor Model library and it uses a special Java Virtual Machine, called JIKESRVM, to manage references invalidation.

Our idea of integrating Parallel Pattern with the Actor Model follows the front of the aforementioned works, which adds abstraction to the Actor Model to improve some peculiar aspects. Differently from them, we mainly focus on improving performance and programmability both in data-parallel and streaming applications.

3.1.5 *Actor Model Benchmarks*

In the fifth and last group of papers, we discuss some performance comparison of Actor Model implementations.

The gold standard of Actors benchmark is the SAVINA benchmark proposed by Imam and Sarkar [126]. SAVINA [188] is a collection of benchmarks taken by different sources and divided into three categories: *micro-benchmarks*, *concurrency benchmarks*, and *parallelism benchmarks*. Both *micro-benchmarks* and *concurrency benchmarks* provide short and light benchmarks aimed to test specific functionalities of Actor Model implementations, e. g., Actor spawn speed, and message communication latency. Instead, the *parallelism benchmarks* provide computationally intensive use-cases such for example the recursive matrix multiplication algorithm. The author initially implemented the SAVINA benchmarks for different Actor Model implementations in the Java ecosystem, e. g., AKKA. Some of the SAVINA benchmarks were then ported to C++ ACTOR FRAMEWORK [219] and to PONY [36, 193] Actor Model implementations.

Blessing et al. [36] criticize the SAVINA benchmarks because they are composed of very different applications that use different paradigms, e. g., Task-Parallelism, thus failing to assess the real performance of practical Actor Model applications. The same authors propose an al-

ternative benchmark based on the implementation of a *chat application* called ChatApp, which, in their opinion, stresses more the asynchronous behavior of Actors. Indeed, the Savina benchmarks include very different kinds of applications, and some do not even respect the Actor Model's constraints making explicit use of shared states [176]. However, Actors are used in many different contexts, and it is essential to have a wide variety of benchmarks that can be useful to compare performance across parallel programming models. In this respect, in Chapter 7 and Chapter 8, we compare C++ Actor Framework with other C++ parallel library considering both data-parallel and streaming benchmarks.

3.2 ACTOR MODEL AS CONCURRENCY MODEL

In this section we present a set of relevant research works, whose primarily focus is comparing the Actor Model with other well-known concurrency models through the analysis of its pros and cons from the Software Engineering standpoint.

Bauer and Makio [31] and Chen et al. [54] use the Actor Model to design applications for the Internet of Things (IoT) and Industry 4.0 domains. The former article designs an interconnected administration shells for the Reference Architecture Model Industry 4.0 (RAMI4.0) [171] in a hybrid cloud environment using Java, and Actor Model and Actor4J. The latter research work uses an Actor-Role-Coordinator (ARC) to model Quality of Service (QoS) requirements in an Open Distributed and Embedded (ODE) system. Both articles agreed that the Actor Model is advantageous to design and implements distributed and heterogeneous applications. The model provides high flexibility while being high-level enough to hide synchronizations and data race issues.

The Actor Model is largely used by programmers to design and implement concurrent applications. However, mainly for the lack of implementations offering the pure Actor Model semantics, its usage does not prevent the insertion of concurrent bugs in the application development process. Tasharofi, Dinges, and Johnson [201] and Swalens et al. [198] criticize in their work the combination of the Actor Model with other concurrency models. The former investigates the motivations that bring Scala programmers to mix the Actor Model with either Task-Parallelism or native Threads. The latter presents an extensive analysis of different combinations of concurrency models, e. g., *Software Transactional Memory*, *Futures/Task-Parallelism*, and Actor Model. The two research works agree on an important point: *combining different models without a suitable high-level abstraction* (like the ones we present in Section 3.1), *can easily bring deadlock and livelock situations that can be challenging to remove.*

Some other works investigate the problems of *deadlock* and *livelock* in the Actor Model [11, 110, 118, 212]. Although Actors do not share state, and the Actor messages are entirely asynchronous, deadlock can still occur if the model is not used correctly, e. g., two Actors waiting for messages from each other [118]. *Deadlock* occurs when Actors are unable to make progress, whereas *livelock* occurs when Actors make only local progress (e. g., by exchanging messages) but the program does not make any global progress.

The works of Hedden and Zhao [118] and Torres Lopez et al. [212] propose a study of most repetitive bugs in the actor-based applications and libraries. Hedden and Zhao [118] study the bugs present in the implementations of some famous open-source libraries based on the Akka. The study was performed by analyzing the repository and the issues tracker of the libraries. The bugs were divided in three main categories, i. e., *communication*, *coordination*, and *logical*. The results of their analysis show that the most present bugs in the Actor Model are the *coordination* ones. Besides, Actor Model, compared to other similar classification of bugs in the context of cloud-based technologies, have fewer bugs in fault recovery.

Torres Lopez et al. [212] propose a taxonomy of concurrency bugs in Actor Model programs based on literature review. The authors divide the bugs into two main categories, namely *Lack of Progress* and *Message Protocol Violation*. They also discuss two types of *deadlock*, i. e., *communication deadlock* and *behavioral deadlock* depending of the specific type of Actor Model implementation. The *communication deadlock* is specifically common in the *Processes* Actor Model type (e. g., Erlang) where it is present the explicit `receive` primitive (see Section 3.4). In this case, two Actors could be blocked forever waiting for a specific message from each other. Instead, the *Behavioral deadlock* is present in the *Classic Actors* Actor Model and all other Actor Model categories. In this case, two Actors are not blocked, i. e., they could process other incoming messages, but they could not make progress because each of them requires a specific message from the other.

The works of Gkolfi et al. [110] and Albert et al. [11] try to analyze the dynamicity of the Actors communicate in order to track *deadlock* and *livelock* situation. Gkolfi et al. [110] proposes an analysis based on transposing Actor graph of communication in a colored Petri nets. Instead, Albert et al. [11] presents a novel *May-Happen-in-Parallel Analysis* applied to the Actor Model concurrency. This can allow the automatic extraction of the maximal level of parallelism of an application for further performance improvement.

Instead, the paper of Fowler, Lindley, and Wadler [103] compare the Actor abstraction with the Chanel abstraction finding out that theoretically they can be derived one from the other. The works of Cardoso et al. [47] compare the Actor Model model with the agent-

based programming language building a possible set of benchmark to compare their functionality.

Finally, there are a relevant set of works that use model-checking technique in order to formal verify Actor Model properties, e. g., Sirjani and Jaghoori [189], D'Osualdo, Kochems, and Ong [67] and Eckhardt et al. [92].

3.3 ACTIVE OBJECTS RELATED WORKS

The Active Objects model is a pattern of concurrency largely inspired to the Actor Model. The goal of the model, is to decouple the object method invocation from its execution to simplify object accesses [137].

The main improvements introduced by the Active Objects are in the communication mechanisms. The Actor Model relies only on messages for all actions (i. e., data movements, managing events and errors, starting a given computation within an Actor, and sending message replies). This may introduce an increasing overhead given by the messages management and potential confusion for the programmer, which has to deal with different semantics associated with different kinds of messages. The Active Objects model exposes a set of distinct methods to the programmer that have a more clear semantics and are type-checked. Those methods always return a *Future* that will be asynchronously resolved to either a message reply or an error.

The Actor Model gives freedom to the semantics of the messages, providing guarantees that messages will be received at most once. Instead, Active Objects provides an explicit distinction between direct messages, results, and error messages with the guarantee to receive acknowledgment communication. Indeed, for each remote method execution corresponds a *Future*. As discussed by Rouvinez and Sobe [181] the Actor Model *at-most-one* message guarantee is similar to the `UDP` network protocol semantics, and the Active Objects *acknowledgments-based* communication is similar to the `TCP` network protocol semantics. Therefore, as it happens for network protocols, the absence of mandatory *acknowledgments-based* communications may improve flexibility and even performance in some specific scenarios.

Besides, it is worth to point out that some of the peculiar features of the Active Objects model are available in several Actor Model implementations, such as the static check of messages and the request-reply protocol pattern, typically implemented with *Futures* [3].

In the following, we report some notable research works that implements extensions of the Active Objects model or that enrich the model with other concurrency models. Henrio, Huet, and István [119],

3 In C++ ACTOR FRAMEWORK, *Futures* are temporary message handler added at runtime, thus they are executed asynchronously and interleaved with other messages received by the Actor.

proposed the Multi-Active Object model, which extends the Active Objects model, allowing each activity to be multi-threaded. Hains et al. [115, 140], proposed a new model that uses Active Objects to coordinate *Bulk-Synchronous Parallel* (BSP) computations. The BSP model can be used to efficiently program data-parallel computations. However, it imposes limitations in the way computing entities must interact. The authors intended to overcome these limitations by integrating the BSP model with Active Objects to improve its flexibility and use Active Objects to coordinate multiple BSP algorithms. The foundational idea of our research is similar to the ideas proposed by Hains et al. [115]. We use structured parallel programming based on common Parallel Patterns to efficiently implement computations on shared-memory platforms while exploiting the flexibility and memory safety guarantees of the Actor Model.

Another example of combining Task-Parallelism with Active Objects is the framework Ray [153]. Ray is a distributed concurrency framework designed to implement Reinforcement Learning algorithms [196], thus capable to cooperate with the most modern Machine Learning libraries. Ray implements a Task-Parallelism model with *Remote Procedure Calls* and *Futures* in combination with some concept of the Active Objects. The combination of Active Objects with the Task-Parallelism, implemented by the Ray framework, is an interesting proposal that aims to homogenize the two models. Tasks are stateless pure functions whereas Active Objects are stateful object with methods.

Finally, Fernandez-Reyes, Clarke, and McCain [101] proposed an extension of the Active Objects model with the ParT abstraction, capable of running efficient data-parallel computations in a non-blocking fashion. The ParT abstraction follow the same principles of the Haskell's *monad* [17]. Specifically, a collection of values can be *lifted* into a ParT type, then a set of data-parallel operations can be applied to the ParT (e. g., map, reduce, filter) and the final result can be *extracted* from the ParT. Besides, the ParT abstraction can execute multiple dependent ParT in parallel and to use different techniques to stop the execution of functions on those values that are irrelevant for the final result.

3.4 ACTOR MODEL LANGUAGES AND LIBRARIES

In this section, we present a view of the most used and recent Actor Model implementation. Over the years, the Actor Model has had an evolution that led to the creation of a wide variety of implementations. We base our discussion on the paper of De Koster, Van Cutsem, and De Meuter [83] who consider four types of Actor Model, namely *Classic Actors*, *Processes*, *Active Objects* and *Communicating Event-Loops*. We extend this categorization by adding a recently developed *Virtual*

Actors proposed by Microsoft Research [159]. We will discuss some more Actor Model implementations, dividing them into standalone languages and libraries for other languages.

The five categories that we consider are:

1. *Classic Actors*, defines the model by using three fundamental primitives: *create*, *send* and *become*.
2. *Processes*, as suggested by the name, it models a set of processes having the capability to send and receive messages.
3. *Active Objects* category, it defines Actors as Objects, which run within an executor and that can send messages through asynchronous method calls.
4. *Communicating Event-Loops* category, which implements Actors as concurrent entities living in a set of event-loops running in a native executor.
5. *Virtual Actors* category, in which Actors are concurrent entities that are automatically spawned when the first message arrives and then remain alive as long as they continue receiving messages.

Table 1 shows the most widespread Actor-based programming languages divided in the first four categories described previously. To the best or our knowledge, there are not standalone languages that implement the *Virtual Actors* paradigm. The table shows the programming paradigm followed by the languages, which can be *Functional* or *Imperative*. We consider as *Functional* the ones that strictly use functional abstraction like first-class functions, higher-order functions, pattern matching, e. g., ERLANG, PONY. Instead, the *Imperative* languages those that focus more on imperative control structures and functions with side effects, e. g., D, DART.

Table 2 shows libraries that implements the different aforementioned versions of Actor Model. The *Language* column reports the programming languages or the software platform that the developers should use to interact with such libraries. Moreover, both tables show the *Last release* date and if they provide out-of-the-box supports in managing Actors on multiple hosts (*Distributed* column).

The languages and the libraries that are grouped in one of the aforementioned categories usually implements all or a subset of the main features of the Actor Model similarly. In the remaining of this section, we will discuss how those categories implements *state management*, *message communication* and *Actor execution*.

3.4.1 *State management*

One of the Actor Model most important features is its ability to update the Actor's internal state during message processing. In the *Clas-*

Name	Last Release	Distributed	Paradigm
Classic Actors			
ACT [142]	1981	_	Functional
Rosette [209]	1997	yes	Functional
Processes			
D [66]	2020	no	Imperative
Elixir [94]	2020	yes	Functional
Erlang [96]	2020	yes	Functional
Active Objects			
ABCL/1 [2]	1990	_	Imperative
SALSA [182]	2011	_	Imperative
Encore [204]	2016	no	Functional
Pony [165]	2020	no	Functional
Communicating Event-Loops			
AmbientTalk [19]	2011	yes	Imperative
E [90]	2016	yes	Imperative
Dart [74]	2020	no	Imperative

Table 1: Actor Model Programming Languages.

sic Actors languages, which follow the theoretical formulation of the Actor Model, the state changes are executed by supplying new behaviors that include the updated state (typically by using the `become` language primitive). In contrast, Actor Languages following the Object-Oriented paradigm and the vast majority of Actor-based libraries represent the state as an object that the Actor can manipulate.

Another crucial property of the Actor Model is the so-called *Actors isolation*, which forces Actors not to share anything. Actor-based programming languages have the advantage of supporting *Actors isolation* natively by restricting at language level the possibility of accessing shared global state or sending references to objects to other Actors. Instead, Actor-based libraries, depending on the programming language in which they were implemented, struggle to restrict the possibility of sharing data structures among Actors. In this respect, since on shared-memory platforms the possibility of using the physical shared memory have advantages, there are some new proposals that uses a *capability-based system* to support *Actors isolation* and to enable safe data sharing between Actors without breaking that important rule [124], e. g., Pony [165] and Encore [204].

Although the *Actors isolation* property is challenging to enforce in Actor-based libraries, some interesting strategies and approaches worth mentioning have been proposed to provide the user with such an important feature. Kilim [131] is a Java-based library that post-processes Java byte-code forcing the deep-copy of each data sent

Name	Last Release	Distributed	Language
Classic Actors			
Akka.NET [9]	2018	yes	.NET (C#, F#)
ProtoActor [169]	2018	yes	.NET, Go, Java, Python, Node.js
C++ Actor Framework [46]	2020	yes	C++
Akka [9]	2019	yes	Java
Actor4j [5]	2020	yes	Java
Nact [154]	2018	no	Node.js (JavaScript, Reason)
Pykka [170]	2019	no	Python
Thespian [207]	2020	yes	Python
Riker [173]	2019	no	Rust
Processes			
F# MailboxProcessor [100]	2019	no	.NET (F#)
Kilim [131]	2018	no	Java
Bastion [30]	2020	yes	Rust
Scala Actor Library [206]	2020	no	Scala
Active Objects			
Actr [6]	2019	no	Java
Comedy [61]	2019	yes	Node.js (JavaScript)
Communicating Event-Loops			
Rotor [63]	2019	no	C++
Actix [4]	2019	no	Rust
Virtual Actors			
Orleans [160]	2019	yes	.NET (C#)
Orbit [158]	2020	yes	JVM (Kotlin, Java)
Acteur [3]	2020	no	Rust

Table 2: Actor Model Libraries.

between Actors. Another approach is the one followed by the C++ ACTOR FRAMEWORK (CAF) that employs the *copy-on-write* method for those messages sent to multiple Actors [46]. The message is not copied as long as none of the destination Actors modify the message. Finally, the RUST language [132] has a linear type system that can be used to track variable ownership. It can avoid mutable accesses on the same variable by distinct threads of control. This RUST features is used in some Actor-based libraries implementations [3, 4, 30].

In principle, the approaches mentioned above may be used to ensure *Actor isolation* in Actor-based libraries. However, since Actors are implemented using general-purpose languages (i. e., JAVA, C++, RUST), the users can always find loopholes to break *Actor isolation* (e. g., by defining shared global variable in JAVA and C++, and by using the `unsafe` block in RUST), thus potentially introducing race conditions among Actors, which instead are prevented by design if the *Actor isolation* property is respected.

3.4.2 *communication mechanisms*

Different Actor Model implementations use different mechanisms to send messages to Actors.

We may distinguish different approaches on the basis of how messages are sent, how messages are received and how they are processed.

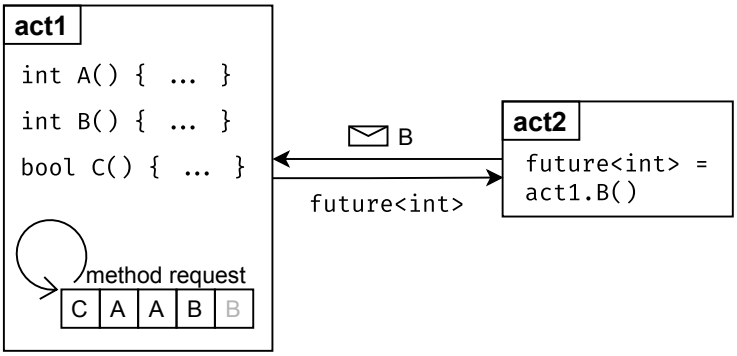


Figure 7: Actors communication through *asynchronous method calls*. `act2` calls a method of `act1` and receives a *Future* of type int.

Concerning the sender Actor, messages can be sent in two ways: by using an explicit *send* primitives like in *Classic Actors* and *Processes*, or by employing *asynchronous method calls* like in the *Active Objects*. *Communicating Event-Loops* and *Virtual Actors* use either *send* primitives or *asynchronous method calls* depending on the implementation (e. g., the *Communicating Event-Loops* language E [90] and the *Virtual Actors* library Orleans [160] use method calls, whereas the *Communicating Event-Loops* library Actix [4] and *Virtual Actors* library Acteur [3] use a send function). The former approach is straightforward. The sending operation is a function that takes as parameter the address of the receiving Actor and the message to send, and then the message will be stored in the mailbox of the destination Actor. Instead, the latter approach leverages the idea of executing a method on the objects (called *Active*) associated with the receiving Actors. The method call enqueues the request of the method execution and immediately returns a *Future*. The *Future*, which is an object used to monitor the state of an asynchronous request, will store either the status of the method invocation (e. g., error or success) or the value result, if any (see also the logical schema of this approach in Figure 7). The two approaches have both advantages and disadvantages that we address in Section 3.3. However, for the sake of technical comparison of Actor Model implementations, it is essential to point out that sending messages through *asynchronous method calls*, from one side reduces the dynamicity of message exchanges, and from the other side, enables compile-time checking of the message types. Notwithstanding, both ways of sending messages should be non-blocking operations.

The most popular Actors implementations (e. g., *Classic Actors*, and *Processes*) usually implement unbounded capacity mailboxes, while a few others (e. g., some *Active Objects* implementations, and *Communicating Event-Loops* library ACTIX [4]) use bounded capacity mailboxes with non-blocking failure in case of the destination mailbox is full. Moreover, Actor Model languages sometimes use some specific syntax for implementing the send primitive. One notable example is the "bang operator" ('`!`') of ERLANG [96] (e. g., `myact ! "hello world"`).

The AKKA [18] Actor Model library has a peculiar feature for what concerns Actor's mailbox. The programmer can choose the type of queue to use for the mailbox of each Actor [144]. There is a set of default queues with different features, e. g., bounded or unbounded queue, priority-based queues, and it is even possible to supply a custom queue. Moreover, AKKA provides mechanisms to replicate Actors by using `Routers` [55]. Messages will be distributed to the replicas according to some pre-defined distribution policies. The `Router` is a simple mechanism to enforce a structure to the topology of Actors in an application. Others Actor Model implementation libraries, e. g., C++ ACTOR FRAMEWORK, has a similar feature [113] called the *Actor Pool*. However, AKKA `Router` provides a set of much more sophisticated distribution policies, one of them being the possibility to send the message to the Actors with the smallest queue length. This feature is supported regardless of the kind of mailbox used by the individual Actor.

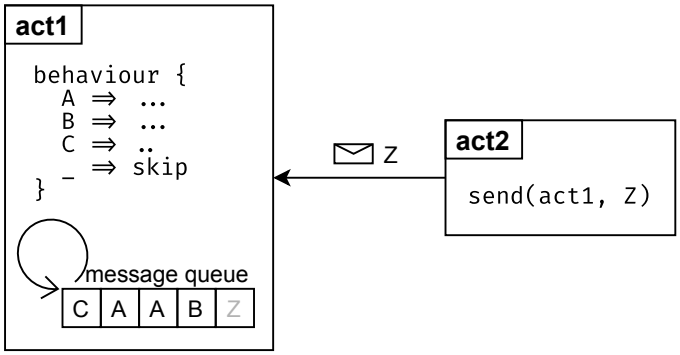


Figure 8: Actors communication through explicit `send` and *Actor behavior*.The Actor `act1` defines a different *behaviors* for each type of input message. Not type-matching messages are left on the mailbox.

Regarding the receiving side of Actor communications, we may distinguish three possible approaches: *Actor behavior*, explicit *receive* primitive/function, and object method definition. We have already discussed how all *Active Objects* and some *Communicating Event-Loops* and *Virtual Actors* implementations use asynchronous method calls as a communication mechanism (see Figure 7). On the contrary, the *Classic Actors* use the *Actor behavior* and *Processes* use an explicit *re-*

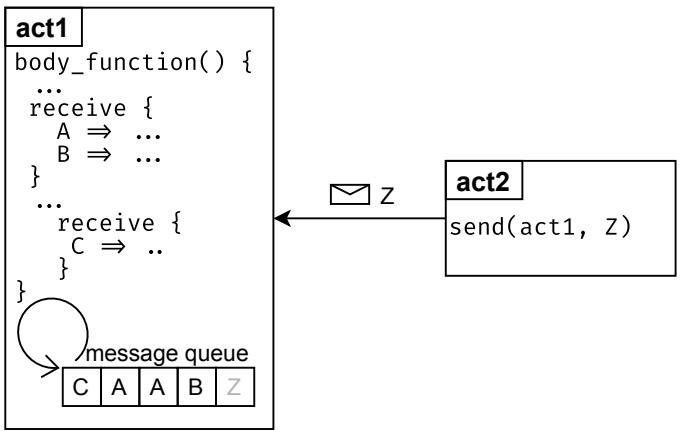


Figure 9: Actors communication through explicit `send` and `receive` primitives. The Actor `act1` executes a body function which has inside multiple *receive* statements. Not type-matching messages are left on the mailbox.

ceive primitive/function. The *Actor behavior* (see Figure 8) is a list of alternative behaviors that are executed based on the type of the received message. The *Actor behavior* can also be changed at run-time with the *become* primitive/function, which gets as a parameter the new behavior to implement. An important characteristic of this kind of messaging system implementation is that the Actors cannot choose to be exempted from receiving messages. Indeed, the selected behavior is executed anyway as soon as a message of that given type arrives at the Actor.

The explicit *receive* primitive/function approach (see Figure 9) permits to wait for messages in every part of the Actor body function. The *receive* operation is usually implemented asynchronously. Therefore if no message is present in the mailbox, the Actor is suspended. This *receive* function returns the next message in the mailbox queue, or in some implementation (e. g., Erlang [96] and Elixir [94]) it returns the message that matches a given pattern. Differently from the object method call approach, both *Actor behavior* and explicit *receive* approaches permit changing which messages can be received dynamically. For this reason, those implementations usually need a more elaborated mailbox queue capable of skipping messages that are not matching the current active handler, but that could be handled in the future. Another possible solution implemented to improve the performance is to drop those not matching messages (e. g., Akka [9], and C++ Actor Framework [46] use this strategy by default). Moreover, the explicit *receive* primitive/function of the *Processes* needs the underline programming language to support asynchronous operations through some forms of resumable green-threads/fibers. This causes difficulties in implementing *Processes* on general purpose languages, thus the *Processes* (as show on Table 1 and Table 2) are usually imple-

mented as a new programming language. Instead, the *Classic Actors*, which does not require those features, is used mainly in library implementations.

Finally, we discuss how messages are processed inside the Actor after a specific handler is selected. The Actor abstraction requires a continuous execution of the handler without interruption that may produce side-effects (e. g., system-calls, or blocking IO operations) to avoid potential deadlocks. Language-based Actor implementations, usually try to avoid interruptions of the Actor execution by using asynchronous function calls. This feature is sometimes available in Actor libraries (e. g., Actor Model library for PYTHON and RUST), but there are no guarantees that the developer will use the non-blocking version. Libraries based on the NODE.JS run-time (e. g., COMEDY [61] and NACT [154]) can leverage the NODE.JS asynchronous implementation to avoid blocking operations and resource sharing.

3.4.3 *Actor execution*

The last Actor Model feature that we consider is how Actors are executed. Actors implementations usually employ the so-called M-N execution model in which M Actors are executed on N native executors (i. e., threads), and usually M >> N [104].

Many *Classic Actors*, *Processes*, and *Virtual Actors* implementations use the M-N execution model with transparent scheduling strategies for Actors, in which each ready Actor is executed on any of the available executors. Usually, this execution model's implementations leverage the Work-Stealing algorithm for balancing Actors' execution on the underlying native executors. There are some notable exceptions, for instance the AMBIENTTALK *Active Objects* programming language uses a dedicated thread of control for each Actor.

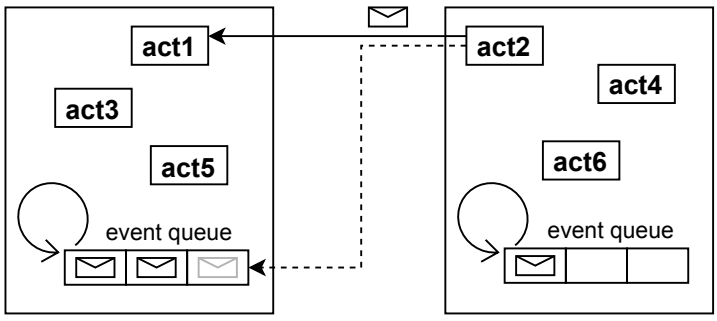


Figure 10: Two communicating event-loops composed of 3 Actors each. The message sent by `act2` to `act1` is enqueued in the event-loop queue and managed concurrently with other events and messages.

Instead, in the *Communicating Event-Loops* category, the programmer has to spawn a set of event-loop executors explicitly, and also has to decide which Actor should run on a given executor. Then, each

Actor will always run on the same executor (see Figure 10). A similar approach is followed when Actor systems provide support for distributed memory platform such as Cloud-based infrastructures. In these cases, the most common approach is to permit the developer to spawn Actors on a specific remote host. Therefore the mapping between Actors and hosts is known to the programmer. However, the *Virtual Actors* implements a different mechanism to make Actors anonymous with respect to the hosts. Actors are spawned as soon as they receive a message in one of the available hosts. The programmer does not know which Actor runs on a specific host, and in case of failure, the next message will be routed on a new instance of the same Actor, potentially running on a different machine.

3.5 DISCUSSION

The Actor Model is a concurrent programming model that has inspired many different research studies. This chapter presents and discusses a broad landscape of Actor Model research works ranging from theoretical studies to more technical proposals. We mainly concentrated on describing efforts that tried to improve or optimize the Actor Model under different aspects. For instance, some works tackle the absence of ways to express structured communication topology [39, 77], and others tackle the problem of implementing efficient synchronization mechanisms [110, 127] or exploiting the shared-memory [81], and more platform-dependent optimizations [105, 123]. The Actor Model issues were faced either by modifying the model or creating a newer enhanced model, but sometimes reducing the Actor Model user-friendliness of Actors [26, 168].

In this thesis, we followed a different path. We proposed a synergic combination of the Actor Model with the Parallel Patterns-based programming approach. The primary aim is to get the best from the two models by maintaining the Actor Model's flexibility and expressivity and bringing platform-specific optimizations through Parallel Patterns. Our research aims to bring Parallel Patterns side-by-side Actors implementing patterns as "macro Actors' to also improve programmability and expressivity.

A related research work that tried to introduce some well-known skeleton implementations of patterns in ERLANG is SKEL [39]. The work primarily aims to improve programmability in the ERLANG Actor-based language by providing ready-to-use parallel pattern implementations. Our approach proposes not only a set of Parallel Patterns as in SKEL, but also aims to define a new synergy between Parallel Patterns and Actors with the objective to enhance both programmability and performance on shared-memory platforms. Our view of Parallel Patterns is that of "macro Actors" that cooperate with standard Actors, according to the Actor Model message-passing

semantics. Indeed, Parallel Patterns have an Actors-based interface, i. e., they can be dynamically spawned, and they have an input mailbox, which are used by standard Actors to communicate with Parallel Patterns.

Others related proposal are CHOCOLA [197] and the PART [101] frameworks, which bring together the Actor Model and the Task-Parallelism model in a single environment. However, the combination of Actors and Tasks has the primary issue that they are two general models with entirely different peculiarities. In practice, their combination does not improve programmability and expressivity, resulting in a more complex programming model for the users. In our proposal, we provide instead a set of well-defined components with their platform-optimized skeletons that efficiently solve recurrent problems in parallel programming without impairing programmability for Actor programmers. The combination of Parallel Patterns and Actors also opens to many opportunities to introduce optimizations within the implementation skeletons transparently, otherwise precluded by the tight Actor Model semantics.

Indeed, the Parallel Patterns allow exploiting shared-memory within the skeleton that implements the pattern for the specific platform (multi-/many-cores in our study). For example, this is the case of patterns solving data-parallel computations (see Chapter 7) or patterns thought for high-throughput Data Stream Processing computations (see Chapter 8). In other research works, the use of shared-memory [81] was provided to Actor-based programmers through the concept of external shared references managed with some restricted policies (e. g., multiple-reader single-writer). On the one hand, these approaches may reduce the message-exchange overhead in situations in which Actors have to share data frequently. On the other hand, it remains challenging and inefficient to implement high-performance data-parallel applications in which multiple Actors need to read and write distinct partitions of some shared data. By using our "macro Actor" approach, this particular usage can be implemented, possibly by using low-level and efficient mechanisms leveraging shared-memory, inside the patterns provided to the Actor programmers for solving data-parallel computations (e. g., *Map*, *MapReduce*, *Stencil*). This is because, inside each Parallel Pattern (i. e., in its implementation skeleton), it is possible, for example, to customize the implementation of mailboxes associated with the Actor entities implementing the skeleton, thus improving the performance of Actors by removing the hidden costs associated to the *memory isolation* principle of the Actors' semantics (see Chapter 5 and Chapter 8).

Finally, there were also attempts to build DataFlow-like computational graphs using Actors, as proposed in Eker and Janneck [93]. Actors are statically connected in a fixed structure. This enables the pos-

sibility to introduce optimizations and improve inter-Actors communications performance thanks to the fixed topology. However, to the best of our knowledge, none of the proposals provide any simple abstractions to the users to build efficient parallel topologies (e. g., parallel pipelines or replication of Actors), thus limiting the programmability and flexibility of these approaches.

4

STATE OF THE ART: PARALLEL PATTERNS

In this chapter, we describe some state-of-the-art solutions in the context of high-level parallel programming, which will help to outline the contributions of this thesis.

Parallelism exploitation is characterized by several complex problems that need to be solved together and at the same time. For example, starting from the sequential problem, it has to be decomposed into several sub-problems, each implemented by a function or by a module. All modules operate in parallel to solve the initial problem. The programmer then has to decide how to map these modules onto some processing elements, select a scheduling policy to balance the workload, and how to hide/avoid costly memory accesses or expensive inter-module communications. Most importantly, those decisions cannot be taken independently, since all the problems mentioned above are reciprocally connected.

One option to deal with these issues and reduce the complexity of the parallelization is to introduce *constraints* in the way the parallel modules are defined and how they can communicate. The aim is to simplify the decisions that have to be made and to enable the possibility to apply well-known heuristics to solve well-structured problems. To this end, one of the most widely acknowledged approaches is to use Parallel Patterns.

4.1 PATTERN-BASE PARALLEL PROGRAMMING

The programming approach based on Parallel Patterns is called *structured parallel programming* [60, 72, 73, 149, 215]. This term has been borrowed from sequential programming where in the 60's and 70's programs were often poorly designed. Several computer scientists recognized that programs should organize code more structurally by using higher-level control structures (e. g., `if-then-else` and `while-do`) aiming to establish structured programming practices [69]. Thus, the programs should be expressed by a composition of a limited amount of abstract high-level constructs.

Structured parallel programming has been envisioned as a viable solution to improve the quality and efficiency of parallel software development while reducing the complexity of program parallelization and enhancing performance portability [148]. Parallel Patterns are schemes of parallel computations that recur in the realization of many algorithms and applications. Based on that, it is possible to build parametric implementations with such well-known paral-

lel structures and rigorous semantics. The user can also decide the best suitable parallel pattern based on a cost model that evaluates their profitability. Each abstraction may be directly instantiated or composed with others to model the complete parallel behavior of the application. This raises the level of abstraction by ensuring that the application programmer does not need to deal with parallelism exploitation issues and low-level architectural details during the application development. Instead, these issues will be efficiently managed using state-of-art techniques by the system programmers who design the structured framework and its associated run-time. This approach liberates the programmers from the concerns of the *process mapping*, *tasks scheduling*, and *load-balancing*, allowing them to concentrate on computational aspects.

Indeed, the programmer can manage a high-level abstract view of the parallel program, while all the most critical implementation choices are in charge of Run-time System that run the Parallel Patterns (e. g., like the *IBMones* proposed in Chambers et al. [48], Gedik, Özsema, and Öztürk [109], and Navarro et al. [155]). This last aspect is usually manually enforced in non–pattern-based parallel programming models such as MPI and PThreads.

More recently, some authors argue that parallel patterns should be used to replace explicit thread programming to improve the maintainability of software [149]. This idea was originated in the late '80s when *algorithmic skeletons* were introduced to simplify parallel programming in the HPC domain [59, 161].

There are some controversies on using the terms Parallel Patterns *and* Skeletons. *In this thesis, we call* Parallel Patterns *the abstract parallel schema and* Skeletons *the actual implementation on a given platform.*

Algorithmic Skeletons (or just "Skeletons") [60] were developed independently of Parallel Patterns to support programmers with the provisioning of standard programming language constructs that model and implement common, parametric, and reusable parallel schemes. Skeletons are pioneered by Cole [59], which defined them as *higher order functions* each of which describes the structure of a particular style of an algorithm. Thus, Skeletons may be considered as a practice implementation of parallel design patterns.

Combinations of parallel design patterns and algorithmic skeletons are used in different parallel programming frameworks such as SkePU [99], Muesli [97], FastFlow [71], and SkeTo [147], Skandium [141], just to mention a few of them.

Other widely popular parallel frameworks such as Google MapReduce [87] and IntelTBB [172] borrowed some Parallel Pattern concepts. Google MapReduce built around a single pattern that strongly limits the user expressibility but provides a powerful tool with highly efficient implementation. Instead, IntelTBB is a parallel library that supports task-based parallel programming and provides some parallel patterns including *parallel pipeline*, *parallel-for*, *parallel-reduce*, and *task-graph*.

4.2 PIONEER SKELETON-BASED FRAMEWORKS

Algorithmic Skeletons (or simply Skeletons) abstract commonly used schemes of parallel computations. Each one provides the user with strict management of process creation, communication, synchronization, and inter- and intra-process data exchange [111]. Skeleton frameworks provide parallel schemas with generic parallel features, which can be parameterized by the programmer to generate a specific parallel program. Skeletons could be instantiated through specific language syntax or library APIs. However, several Skeleton frameworks offer custom language syntax to take full advantage of the abstractions provided as well as to target different platforms with the same high-level code. Some examples were *Structured Coordination Language* (SCL) [72], *Pisa Parallel Programming Language* (P^3L) [28], and *Skeleton-based Integrated Environment* (SkIE) [27].

SCL was one of the first languages introduced for Skeletal programming. It provided a base language that is designed to be integrated with a host language (e. g., Fortran). In SCL, Skeletons was classified into three types: *configuration*, *elementary*, and *computation*. Elementary Skeletons and Computation Skeletons was respectivelly Data-parallel Skeletons (e. g., *map*, *scan*, and *fold*) and Task-parallel Skeletons, (e. g., *farm*, *SPMD*, and *iterateUntil*). Instead, Configuration Skeletons abstract commonly used data structures such as distributed arrays (*ParArray*). SCL skeletons were instantiated in Fortran, but SCL needed an additional compilation layer to produce the final code.

P^3L was a skeleton-based coordination language. It provided skeleton constructs that coordinate the parallel or sequential execution of C code. P^3L used implementation templates to compile P^3L code into a target architecture. Thus, a Skeleton could have several templates, each optimized for a different architecture, and provided a parametric process graph with a performance model. A P^3L module corresponded to an adequately defined Skeleton construct with input and output streams, and other submodules or sequential C code. Modules could be partially nested using a two-tier model. The outer level is composed only of task-parallel skeletons, while data-parallel ones could be used in the inner levels. P^3L also supported type verification, and in this case, the programmer had to explicitly specify the type of the input and output streams and the type of each submodule and C code.

SkIE was similar to P^3L. It provided advanced features such as debugging tools, performance analysis, visualization, and graphical user interface. Indeed, programmers could interact with a graphical tool, where parallel modules based on skeletons could be composed in the P^3L style.

4.3 PARALLEL PATTERNS LIBRARIES

In this section we present the most relevant Parallel Pattern libraries that are currently maintained and updated, i. e., MUESLI, FASTFLOW, SKEPU, and GRPPI.

4.3.1 MUESLI *Skeleton Library*

MUESLI [134], developed at University of Müsnster, is a C++ template library that supports both shared-memory as well as distributed memory architectures. It uses MPI for inter-node communications and OPENMP for intra-node parallelism exploitation. Recently, it has also been extended to support multi-GPU systems by using CUDA [97]. It provides data-parallel skeletons such as *Map*, *Fold*, *Scan* (i. e. prefix sum) *zip* and *mapStencil*. MUESLI also implements distributed data structures such as distributed arrays, matrices, and sparse matrices (in the current version, skeletons operating on sparse matrices cannot be executed on GPUs). Data parallel skeletons are offered to the user as member functions of distributed data structures. Listing 3 shows a Muesli implementation of the Frobenius Norm computation for a matrix of size `dim` × `dim` using the `Map` and `Fold` skeletons.

```
1  int main(int argc, char** argv) {
2     // initialize Muesli
3     msl::initSkeletons(argc, argv);
4     // create and initialize a distributed matrix dim x dim
5     msl::DMatrix<float> A(dim, dim, 1, msl::Muesli::num_total_procs,
6        [](int row, int col){
7           return randomFloat(row, col);
8     }, Distribution::DIST);
9     // create user functions
10    auto square = [](float a) {return a*a;};
11    auto sum = [](float a, float b) {return a+b;};
12    // apply skeletons
13    A.mapInPlace(square);
14    float f_norm = A.fold(sum);
15    // write result
16    std::cout << "||A||_F = " << sqrt(f_norm) << std::endl;
17    // terminate Muesli
18    msl::terminateSkeletons();
19 }
```

Listing 3: MUESLI program computing the Frobenius Norm of a matrix.

Task-Parallelism skeletons are offered as separate classes. They are used to construct process topologies such as *Farm*, *Pipeline*, *Divide&Conquer* and *branch-and-bound*. To use a Task-Parallelism skeleton, the user has to instantiate an object of the corresponding class. When nesting distributed data structures into Task-Parallelism skele-

tons, only a subset of processes participate in the Task-Parallelism skeleton. In the current version of MUESLI the programmer must explicitly indicate whether GPUs are to be used for data-parallel skeletons.

4.3.2 *Generic and reusable Parallel Pattern Interface (GrPPI)*

GRPPI is a programming interface for modern C++ applications developed at the University Carlos III of Madrid [44, 179]. It is an open-source library that accommodates a layer between application developers and existing parallel programming frameworks targeting multi-core systems. According to GRPPI authors, the lack of common terminology to denote different kinds of patterns and the lack of a recognized and assessed API to use these patterns has prevented the wide diffusion of the pattern-based parallel programming as well as a general acknowledgment of the related advantages. By using advanced C++ features such as meta-programming concepts, and generic programming techniques, GRPPI provides a fully C++ compliant API to well-known Parallel Patterns on top of different programming models offering a unified standard interface. GRPPI currently supports as run-times ISO C++ Threads, OPENMP, INTELTBB, and recently FASTFLOW [106].

GRPPI provides four main components: i) type traits, ii) pattern classes, iii) pattern interfaces, and iv) execution policies. Type traits are used for function overloading and to allow compositions of different patterns. Moreover, by using the `enable_if` type trait from the C++ standard library, functions that are not used are removed at compile time so that only functions meeting the conditions become available to the compiler, thus allowing to provide the same interface for different implementations.

GRPPI provides a set of independent classes that represent each of the supported patterns. These objects store references to other functions and non-functional information (e. g., concurrency degree) related to the pattern configurations. Thus, they can be used inside another pattern to express complex constructions that can not be represented by leveraging a single pattern.

For each supported parallel pattern, GRPPI offers two different alternatives, one for pattern execution and one for composition with other patterns. Both alternatives receive the user functions that will be executed accordingly to the pattern semantics, and configuration parameters as function arguments. Listing 4 shows a simple GRPPI pipeline computing $F(G(x))$ over a stream of N elements generated by the first stage.

Finally, a key point of GRPPI is the ability to easily switch between different programming framework implementations of the same pattern. This is achieved by providing a set class that encapsulates the

```
1  void exec_pipeline(grppi::polymorphic_execution& e, int N) {
2     grppi::pipeline(e,
3        [x=0.0,N]() mutable -> std::optional<double> {
4           if (x < N) return x++; // produce an element
5           else return {}; // end-of-stream
6        },
7        [](double x) { return F(x); }, // apply F
8        [](double x) { return G(X); }, // apply G
9        [](double x) { std::cout << x << std::endl; } // print result
10    );
11 }
```

Listing 4: GrPPI pipeline computing $F(G(x))$ for a stream of N elements.

actual pattern implementations of a given framework. The result is that it is straightforward to use different RTS frameworks. The current GrPPI version provides support for sequential, C++ threads, OpenMP, IntelTBB, and FastFlow frameworks.

4.3.3 FastFlow

FastFlow is a C++ parallel programming library targeting multi/many-cores and offering a multi-level API to the parallel programmer [16, 71, 210]. At the top level of the FastFlow software stack, there are some ready-to-use high-level parallel patterns such as *Pipeline Task-Farm*, *ParallelFor*, *Divide&Conquer*, *StencilReduce*, *Macro Data-Flow*. At a lower level of abstraction, the library provides customizable sequential and parallel *building blocks* addressing the needs of the run-time system programmer (see Figure 11). The idea is that new high-level patterns or new high-level libraries and Domain Specific Languages (DSLs) can be built by a proper assembly and nesting of the *building blocks* [13, 210].

The FastFlow library is open-source under the LGPLv3 licence[1] and realized as a modern C++ header-only template library. The library was conceived to support highly efficient stream parallel computations on heterogeneous multi-/many-cores. The programmers define their parallel applications as a structured directed data-flow graph (called *concurrency graph*) of processing *nodes*. A FastFlow *node* represents a basic unit of computation. Each *node* can have zero or more input channels and zero or more output channels. The graph of concurrent nodes is constructed by the assembly of sequential and parallel *building blocks* as well as higher-level parallel patterns. A generic node of the concurrency graph (being it either standalone or part of a more complex parallel pattern) performs a loop that: i) gets a data item (through a memory reference to a data structure) from one of its input channels; ii) executes a functional code (i. e., business

1 FastFlow home: http://calvados.di.unipi.it/fastflow

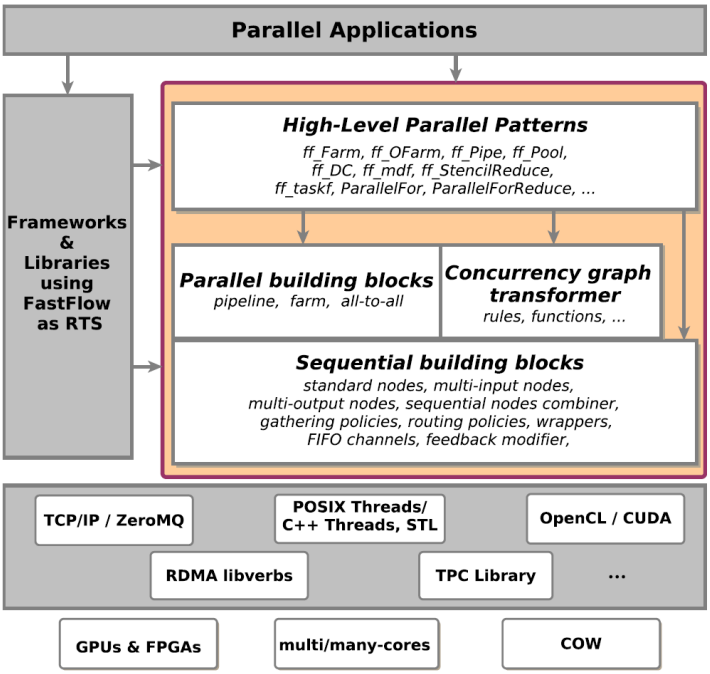


Figure 11: The FastFlow-3 software layers.

logic) working on the data item and possibly on a state maintained by the node itself; iii) puts a memory reference to the result item into one or multiple output channels selected according to a predefined or user-defined policy. Input and output channels are implemented with a Single-Producer Single-Consumer (SPSC) FIFO queue. Operations on FastFlow queues (that can have either bounded or unbounded capacity) are based on non-blocking lock-free synchronizations enabling fast data processing in high-throughput streaming applications [15].

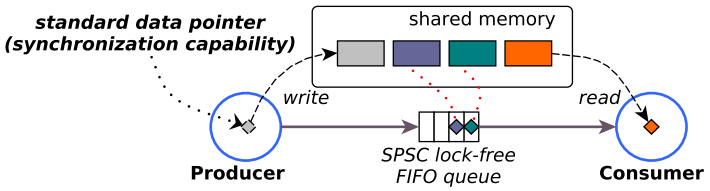


Figure 12: FastFlow library producer-consumer semantics: sending references to shared data over a SPSC lock-free FIFO channel.

From the programming model standpoint, the FastFlow library follows the well-known Data-Flow parallel model where channels do not carry plain data but references to heap-allocated data. The semantics of sending data references over a communication channel is that of transferring the ownership of the data pointed by the reference from the sender node (producer) to the receiver node (consumer) (i. e., Figure 12). The data reference is *de facto* a *capability*, i. e., a logical token that grants access to a given data structure or to a portion of a

data structure. On the basis of this *reference-passing* semantics, the receiver is expected to have exclusive access to the data value received from one of the input channels, while the producer is expected to not use the reference anymore. This semantics is not directly enforced by the library itself with any static or run-time checks.

FastFlow-3 [210] introduces the new concepts of *building blocks* and automatic graph transformation, which improve both performance and flexibility. *Building blocks* are concurrent components that are the fundamental elements of any structured parallel applications implemented using the FastFlow library. Building blocks are either sequential or parallel. Sequential *building blocks* are FastFlow nodes with one or more input or output channels. Parallel *building blocks* are concurrent components made out of a proper assembly of multiple nodes and multiple SPSC FIFO channels. FastFlow proposed three parallel *building blocks* (two of them were included in the previous FastFlow versions as core patterns), *Pipeline*, *Farm* and *all-to-all*, which can be specialized in different ways using also the feedback channel modifier (see Figure 13).

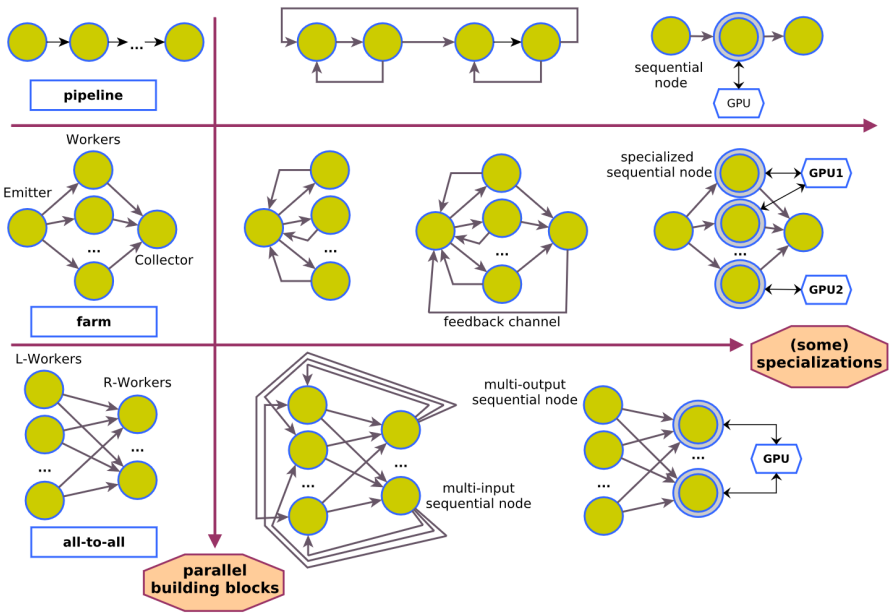


Figure 13: FastFlow parallel *building blocks* and some possible specializations of them.

The *Pipeline* is used both for connecting *building blocks* and to express data-flow *Pipeline* parallelism at run-time. The *Farm* models functional replications coordinated by a centralized `Emitter` entity and a centralized `Collector` entity (that might not be present), which can be specialized by the user to define custom data distribution and data gathering policies. The *all-to-all building block* models both functional replication without a centralized coordination entity as well as the shuffle communication pattern between function replicas. It al-

lows to remove potential bottlenecks in the topology introduced by the *Farm building block* having one or two centralized entities (i. e., the `Emitter` and the `Collector`). The *all-to-all* also enables the fusion operation of two (or more) farms in a pipeline. From the programmer perspective, the reduced set of sequential and parallel *building blocks* with their customizability and composability features, enables the so-called LEGO-style approach to parallel programming where the "bricks" can be either complex pre-assembled and already tested structures or elementary sequential and parallel *building blocks*.

```
1  struct Source: ff_node_t<float> {
2     Source(std::vector<float>&D) : D(D) {}
3     float* svc(float *) {
4        for(size_t i=0; i<D.size(); ++i)
5           ff_send_out(&D[i]); // streaming elements
6        return EOS; // End-Of-Stream
7     }
8     std::vector<float>& D;
9  };
10 struct Worker : ff_node_t<float> {
11    float* svc(float* in) { return F(*in); }
12 };
13 struct Sink : ff_node_t<float> {
14    float* svc(float *in) { sum += *in; return GO_ON; }
15    float sum = 0.0;
16 };
17 int main(int argc, char *argv[]) {
18    // ...
19    Source S(D);        // Source object
20    std::vector<ff_node*> W; // pool of Workers
21    for(size_t i=0; i<4; ++i)
22       W.push_back(new Worker());
23    Sink R;             // Sink object
24    ff_farm farm(W, S, R); // farm building block
25    farm.run_and_wait_end(); // run and wait for termination
26    // ...
27 }
```

Listing 5: A simple FastFlow example that uses the ff_farm building block.

Listing 5 shows a basic FastFlow example that computes $\sum_i^N F(D[i])$ where D is a vector of size N. It uses the *Farm* building block. All data elements of the array D are streamed toward the pool of farm `Workers` by the farm `Emitter` (in the example only 4 `Workers` are used). The reduction phase (i. e., the summation of all results) is computed sequentially by the farm `Collector` and in pipeline fashion with the `Emitter` and the pool of `Workers`. Basically, the farm *building blocks* semantically equivalent to a three-stage pipeline whose middle stage is replicated a given number of times. The first node, which produces the stream of data, is defined at line Line 1 and it is instantiated at Line 19. It generates a stream of N elements (`N=D.size()`) by using the method `ff_send_out` (Line 5) and then it generates the

`EOS` special value (*End-Of-Stream*), which allows the RTS to start the termination phase. From Line 20 to Line 20 an `STL` vector containing 4 replicas of the Worker node is created. The Worker is defined at Line 10. Finally, the `Sink` node is defined at Line 13 and instantiated at Line 23. It collects all results produced by farm `Workers` and accumulates partial results in the `sum` local variable. The farm object uses as `Emitter` the `Source` node and as `Collector` the `Sink` node. It is created at Line 24 by passing as arguments of the `ff_farm` constructor the vector containing the Worker replicas and the objects implementing the `Source` and `Sink` nodes.

4.3.4 SkePU *Programming Framework*

SkePU [95, 98, 99] is an open-source high-level C++ programming framework for heterogeneous parallel systems, with a primary focus on multi-core CPUs and multi-GPU systems. Its main objectives are to enhance both performance portability and to provide a more programmer-friendly interface than low-level APIs such as OpenCL and CUDA. It is implemented as a C++ template library that provides a unified interface for data-parallel computations through algorithmic skeletons both on GPUs using OpenCL and CUDA backends, multi-core CPUs by using a parallel OpenMP backend and cluster of workstations by using MPI.

The first version of SkePU (called SkePU-1), developed until 2015 (the latest release was SkePU 3.0), used a macro-based language where C preprocessor macros were used to abstract the target platform. The SkePU-1 user functions, generated from a macro interface, were C++ objects containing member functions for CUDA and CPU targets, and strings of code for the OpenCL target to be dynamically compiled. Deciding which backend to use for a given application depends on several different factors such as the problem size, the kind of target platforms, and the skeleton used. Dastgeer, Li, and Kessler [76], the SkePU authors, proposed an automatic selection algorithm based on an offline machine learning algorithm that generates a decision tree with low training overhead end provides the user with an auto-tuning mechanism for backend selection.

SkePU-1 [95] includes different flavors of the Map and Map-Reduce skeletons (`Map`, `MapArray`, `MapOverlap`, `Reduce`, `MapReduce`), the `Scan` skeleton that is a generalized prefix sum operation with a binary associative operator, and the `Generate` skeleton that allows us to initialize elements of an aggregate data structure based on the element index and a shared initial value. SkePU-1 includes also aggregate data structures called smart containers [75]. Smart containers are concurrent data structures of generic elements (currently available as vectors, matrices, and tensors) stored in the host's main memory, but that can temporarily store subsets of their elements in GPU device memories

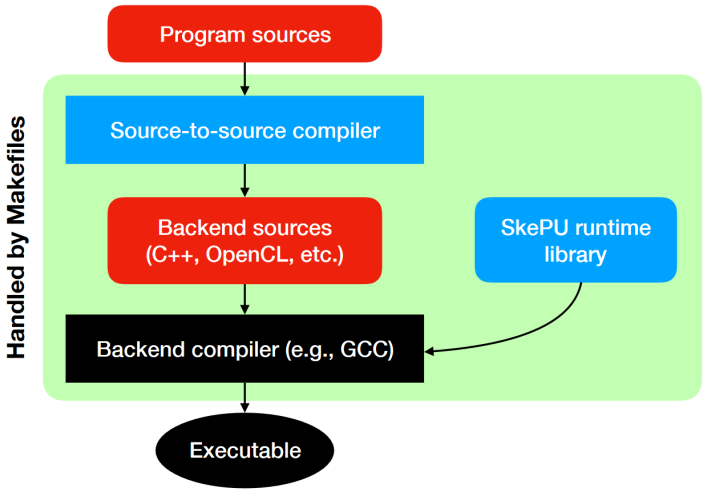


Figure 14: The SkePU compiler infrastructure.

to optimize memory-to-memory transfers and device memory allocation. Besides, smart containers perform transparent software caching of kernel operands they wrap.

SkePU-2 [99] is a redesign of SkePU-1 made available in 2016. It builds on the run-time system of SkePU-1 updated to use C++ variadic template features. It also adds a new user interface based on the modern C++ syntax that leverages lambda expression and a new compilation model with a source-to-source translation. While SkePU-1 uses preprocessor macros to transform user functions for parallel backends, SkePU-2 utilizes a source-to-source precompilation phase based on libraries from the Clang project [205]. The user source code is provided through this tool before the standard compilation phases (see Figure 14). The main aim of the redesign of SkePU-1 is to enhance flexibility and type-safety by removing macros and leveraging C++ features for cleaning up the user interface. SkePU-2 removes also the `MapArray` and the `Generate` skeletons in favor of a generalized `Map` skeleton and adds the new `Call` skeleton.

SkePU-3 [98] is the new iteration of the framework, which focuses on improving flexibility and programmability of the framework along with a new MPI-based backend. The author provides two new variants of the `Map` and `MapReduce` skeletons, respectively the `MapPairs` and `MapPairsReduce`, which compute 2D domain matrix from two sets of 1D vectors. The `MapPairs` applies a cartesian product-style pattern from two `Vector<T>` sets. Each Cartesian combination of vector set indices generates one user function invocation, the result of which is an element in a Matrix. The `MapPairsReduce` is the combination of a `MapPairs` followed by a row-wise or column-wise reduction over the generated matrix elements, which results in a column or row vector.

Two new container are provided, the `Tensors` container and `MatRow` container proxy. The former provides 3 and 4 dimension structures

to complement the `vector` and `matrix` container. The latter is a proxy container to apply some computation to a specific row of a matrix. Finally, SkePU-3 adds dynamic scheduling in the OpenMP and a new memory model.

```
1  // map function working on the single item of the input collection
2  MapOutput mapFunction(skepu::Index1D index, parm elem)
3  { /* business-logic code */ }
4  // preparing the input and output data structures
5  skepu::Vector<parm> swaptions_sk(swaptions, nSwaptions, false);
6  skepu::Vector<MapOutput> output_sk(nSwaptions);
7  // creating the map object by providing the function to compute
8  auto map = skepu::Map<1>(mapFunction);
9  // setting up the OpenMP backend and the number of threads to use
10 auto spec = skepu::BackendSpec{skepu::Backend::Type::OpenMP};
11 spec.setCPUThreads(nThreads);
12 map.setBackend(spec);
13 // map execution invocation
14 map(output_sk, swaptions_sk);
```

Listing 6: SkePU implementation of the *Swaptions* Parsec benchmark using the OpenMP backend.

An example of SkePU code is shown in Listing 6 where a single *Map* pattern is used to parallelize the main kernel of the *Swaptions* Parsec application. The function containing the business logic code operating on each input data element is defined in Line 2. Input and output data collections are instantiated by using SkePU smart containers (Line 5 and 6 respectively). Then, the map object is created by providing the map function (Line 8), and the OpenMP backend run-time (with its parallelism degree, Line 11) is selected for the map pattern at Line 10. Finally, the data-parallel computation is executed at Line 12.

4.4 DISCUSSION

Parallel programming based on the Parallel Patterns is a well-acknowledged approach for parallelizing applications. Parallel Patterns are high-level parallel abstractions, each specifically designed to solve a recurrent problem efficiently. They enable the possibility to introduce platform-specific optimizations within the implementation skeletons of the patterns. Patterns have clear parallel semantics and straightforward APIs for the user. All implementation details and platform-specific optimizations for a given pattern are transparent to the user.

Parallel Patterns were already proved to achieve performance comparable to manual implementations of some Parsec applications with a significant reduction of the programming effort while maintaining

performance portability across multiple heterogeneous multi-/many-core platforms [85].

However, since the first introduction of skeleton-based libraries, one of the downsides of the pattern-based approach is its flexibility. That is, the possibility to use Parallel Patterns within already written and not well-organized applications and the difficulty of having ready-to-use patterns for the many possible different problems encountered in real-life applications.

Moderns Parallel Pattern frameworks, like GrPPI [44] and SkePU [98], partially solve these issues by provides a library atop general-purpose languages such as C++. FastFlow recently introduced a building block software layer within the library [210], which gives the programmers the possibility to quickly build the patterns they need, whether the ones already provided by the library are not suitable for the problem at hand. Building blocks also overcome the flexibility limitations of the pattern-based approach, even though they have a lower abstraction level than Parallel Patterns, and they are more suited to be used by expert programmers to build new RTS for new frameworks or Domain-Specific Language (DSL).

The combination of the Actor Model with Parallel Patterns, which we proposed in this thesis, on the one hand, tries to tackle the flexibility issues of current pattern-based frameworks targeting multi-/many-cores and, on the other hand, brings the possibility to introduce performance optimizations, typically used in the implementation skeletons of Parallel Patterns, within the Actor Model. The resulting model combines both the efficiency and expressivity power of patterns as well as the flexibility, the memory isolation guarantees, and the expressivity power of Actors. The programmers can use Actors to model and build their applications dealing with dynamic spawning of Actors and complex interconnections of Actors if needed. Instead, the performance-critical parts of the application can be designed by using Parallel Patterns and their compositions. Indeed, since we modeled Parallel Patterns as "macro Actors", they can also be dynamically spawned by standard Actors and can communicate with other Actors and other Parallel Patterns only through explicit messages, thus preserving the semantics of the Actor Model.

Part II

TOWARDS A SYNERGIC COMBINATION OF ACTORS AND PARALLEL PATTERNS

In the following Chapters, we analyze the Actor-based programming model considering performance and programmability issues on multi-/many-cores. Then, we present our attempts to build a synergy between the Actor Model and the structured parallel programming approach based on Parallel Patterns.

In Chapter 5, we show the performance limitations of enforcing the memory isolation property on shared-memory platforms.

Chapter 6 we proposed our first attempt to combine Actor Model with Parallel Patterns thought a software artifact capable of accelerating data-parallel computations.

Chapter 7 we design a set of Parallel Patterns that can be combined to implement critical parts of the Actor-based application.

Finally, Chapter 8 shows specialized implementation skeletons of our Parallel Pattern library targeting high-throughput data streaming processing applications.

5

ANALYZING THE ISOLATION PROPERTY ON MULTI-/MANY-CORE PLATFORMS

Parts of this chapter have been published in the Parallel Computing: Technology Trends, Proceedings of the International Conference on Parallel Computing *(ParCo) [174].*

In multi-/many-core systems, the physically shared memory is the primary means of cooperation among threads and processes running on different cores. Communications occur implicitly through loads and stores coordinated by synchronization protocols typically implemented using *locks*. Locks seriously limit concurrency. They are costly operations requiring the intervention from the OS to suspend the thread and restore it later. Moreover, locks might introduce deadlock situations into the application and increase the debugging and maintainability software phases.

A different approach uses the pure message-passing programming model to coordinate the computation of the concurrent entities. In this model, shared states are either partitioned and, in some cases, replicated among available entities so that each one encapsulates and manages a local state. Explicit messages between entities are used to coordinate concurrent accesses to the local state and enable computation progress without extra synchronizations. The message-passing model is the reference model in distributed systems. However, it can also be used as a coordination model in shared-memory platforms [215], provided that each message is self-contained without any reference to global or heap-allocated memory resources. This property is called *memory isolation* or simply *isolation*. Isolation is an important property for guaranteeing memory-safety. It is one of the most important properties of the Actor Model (cf. Section 2.3.1).

Indeed, sharing mutable data references in a producer-consumer fashion is generally more efficient than the explicit sending of data, particularly for large data structures. However, data sharing coordinated by messages is dangerous. A wrong message protocol, and wrong changes to a data reference, might propagate producing unexpected data-races. A data-race occurs when two concurrent operations (where at least one is a write operation) to the same memory location are not correctly synchronized. On the contrary, maintaining memory isolation might require to copy data during the message-passing operations, thus introducing extra overheads that might have a significant impact on the overall application performance. Nevertheless, maintaining the equilibrium between performances and the memory isolation guarantees is crucial in modern parallel programming targeting shared-memory systems.

This chapter analyzes the performance implications of the *isolation* property on multi-/many-core platforms through different message-passing semantics implementations.

We start this chapter by describing the reference platforms we used for the experiments in this and the other chapters of the thesis (Section 5.1). Then, Section 5.2 discusses the importance of having compile-time checks for the isolation property. We also considered the differences in the programming model of the three technologies that we selected (i. e., FastFlow, C++ Actor Framework, and Rust). The chapter continues with the practical implications of using statically enforced isolation property in different kinds of computations. Indeed, we investigate the dataflow computation in Section 5.3 and data-parallel computations in Section 5.4. As we will demonstrate, enforcing isolation in data-parallel computations introduces extra overheads. Such overheads could be avoided by raising the abstraction level and introducing high-level structures (i. e., Parallel Patterns) with well-defined parallel semantics and efficient implementations for the target platforms.

5.1 MULTI-/MANY-CORE PLATFORMS

For the evaluation reported in this and the next chapters, we used four different server platforms representative of current multi-/many-cores technology, i. e., `Xeon`, `KNL`, `Power8`, `Epic`.

Name	Socket	Core		Frequency	Cache			RAM
		Physical	Logical		L1	L2	L3	
Xeon	2	24	48	2.40 GHz	32 kB	256 kB	30 MB	64 GB
KNL	32	64	256	1.3 GHz	32 kB	1 MB	-	96 GB
Power8	2	20	80	3.69 GHz	64 kB	512 kB	8 MB	64 GB
Epic	2	32	64	2.4 GHz	96 kB	512 kB	64 MB	128 GB

Table 3: Technical specifications of the four reference platforms used for the tests.

While Table 3 summarizes the technical specification of the server platforms, in the following we provide a detailed description of each of them:

`Xeon` It is equipped with two *Intel E5-2695* Ivy Bridge CPUs running at 2.40 GHz 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, running *Linux 4.15.0 x86_64* with the CPUfreq performance governor enabled and turbo boost disabled at boot time. Available compiler is the GNU `gcc` version `9.0.1`.

`KNL` It is equipped with a *Intel Xeon Phi 7210* codename Knights Landing (KNL). The KNL has 32 tiles (each with two cores)

working at 1.3 GHz, interconnected by an on-chip mesh network. Each core (4-way Hyper-Threading) has 32 kB L1D private cache and a L2 cache of 1 MB shared with the sibling core on the same tile. 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 3.10.0* and the GNU `gcc` compiler version `7.3.0`.

`Power8` It is a dual-socket IBM server *8247-42L* with two *Power8* processors each with 10 cores organized in two CMPs of 5 cores working at 3.69 GHz. Each core (8-way Hyper-Threading) 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*. Available compiler GNU `gcc` version `8.2.0`.

`Epic` It is equipped with two CPUs *AMD EPYC 7551* and 128 GB of DRR4 RAM. Each CPU has 32 cores (two logical threads) organized in groups of four cores sharing a L3 cache of 8 MB (the total number of logical cores of the machine is 128). Each core has a clock rate of 2.4 GHz, a L1 cache of 96 kB and a L2 caches of 512 kB. The machine runs *Linux 4.15.0 x86_64* and the GNU `gcc` compiler version `9.0.1`.

5.2 THE NEEDS FOR ISOLATION

The message passing semantics built on top of the isolation principle is crucial because it guarantees computation correctness avoiding possible data races, and provides advantages both in terms of clarity and programmability. The programmers can better compose their applications with a clear understanding of the data's ownership and how those data moves from one entity to another. This is a concept shared by multiple parallel programming models based on message passing, but not in all of them the implementation enforces the isolation constraints to the programmer. In the following, we show three technologies, which, although they are based on the different programming models, adopt distinct approaches to isolation enforcement in message passing. In particular, we show FastFlow a dataflow parallel library, C++ Actor Framework an Actor Model implementation, and Rust a system programming language.

FastFlow (cf. Section 4.3.3) is a parallel library offering both high-level parallel patterns as well as composable parallel building blocks. It follows the *Dataflow Model* leveraging high performance message passing semantics. The *Dataflow Model* (cf. Section 2.3) is based on the concepts of having data "flowing" through a graph of concurrent entities that are called *Node*s. The graph's Nodes can be executed con-

currently based on the fact that they are isolated entities, and entities take complete ownership of the data flowing. However, this semantics is not enforced by the FastFlow library, leaving the burden of respecting the semantics directly to the programmer.

```
1  struct Stage1:ff_node_t<float>{
2   Stage1() : base(2*N) {}
3
4   int svc_init() {
5    initialize(base);
6    p1 = base.data();
7    p2 = p1 + N;
8    std::swap(p1, p2);
9    return 0;
10  }
11
12  float* svc(float* in) {
13   if(haveToStop(p1, p2))
14     return EOS;
15   std::swap(p1, p2);
16   ff_send_out(p1);
17   workS1(p2, N, 10.0);
18   return GO_ON;
19  }
20
21  std::vector<float> base;
22  float *p1, *p2;
23 } S1;
24
25 void
26 workS1(float*, size_t, float) {
27  // body function of Stage1
28 }
```

```
29 struct Stage2:ff_node_t<float>{
30  float* svc(float* in) {
31       workS2(in, N, 20.0);
32       return in;
33  }
34 } S2;
35
36 void
37 workS2(float*, size_t, float) {
38  // body function of Stage2
39 }
40
41 int main() {
42  // creates the pipeline
43  ff_Pipe pipe(Stage1, Stage2);
44
45  // creates the feedback channel
46  pipe.wrap_around();
47
48  // synchronous execution
49  if (pipe.run_and_wait_end()<0) {
50    error("running pipe\n");
51    return -1;
52  }
53  return 0;
54 }
```

Listing 2: A simple producer consumer program in FastFlow.

As an example, a valid FastFlow program is the one sketched in Listing 2. It implements a two-stage *Pipeline* where the two stages work disjointly on two distinct portions of the same vector in a producer-consumer fashion. The producer (`S1`) allocates a standard vector of size `2N` (Line 2) and then uses two raw pointers to point to two distinct parts of the vector that are swapped at every producer-consumer iteration (Line 15). Each stage works on a portion of length `N` of the initial vector (Line 17 and Line 17 respectively). The logical schema of this simple producer-consumer use-case is sketched in Figure 15.

In this simple example, there is no guarantee that within the `workS1` or `workS2` functions (Line 27 and Line 38 respectively) some wrong accesses to a portion of the vector may produce data-races due to buffer overruns.

This implementation breaks the isolation property. The Actor Model strongly relies on Actor isolation and some Actor Model

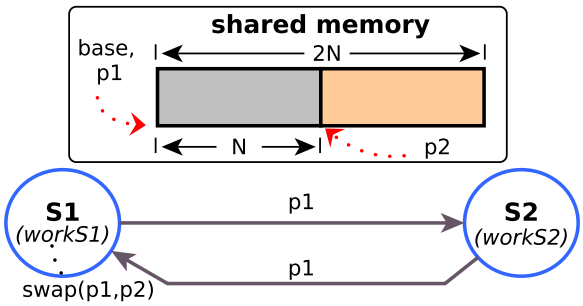


Figure 15: Logical schema of the FastFlow two-stage *Pipeline* described in Listing 2.

implementations such as the C++ Actor Framework (CAF) (cf. Section 2.4.1) uses mechanisms to enforce it. CAF uses modern C++ move semantics and a metaprogramming functionalities for sending messages between Actors. The c++11 standard introduces a new non-const reference type called *rvalue reference*, identified by &&. This refers to temporaries that can be modified after they are initialized to enable "move semantics". Moving an object means copying only the stack-allocated part of the object and move all the data inside via heap-allocated references. The *rvalue reference* permits access to the internal of the object and sets all the pointer to null after the move operation occurred. The standard introduces also the `std::move` function to change a *lvaue reference* to a *rvalue reference* and to facilitate the move semantics usage.

Moreover, there is a recent trend of programming languages that provide strong type systems. For instance, the Rust programming language (cf. Section 2.4.2) uses a capability system to enforce at compile-time the ownership rule. Thus, message-passing libraries built atop Rust could use ownership rules to guarantee memory isolation between the sender and the receiver.

The programmer who wants to implement a similar program using these new languages is forced to declare two separate vectors and alternatively move the vectors' ownership through the communication channels connecting the two nodes. CAF limits the possibility to share pointers between Actors at the level of the library. The library cannot permit sending a pointer to other Actors. It is allowed only to move or copy an entire message. Rust, instead, statically enforces the ownership rule, which, in this case, is violated by the concurrent ownership of the vector by the two pipeline stages. Moreover, memory accesses outside the boundaries of the two vectors are checked at run-time.

The two approaches achieve the same objective in two different ways, CAF uses C++ metaprogramming facilities to force the user to maintain isolation between Actors. However, this approach is not perfect because it can easily be bypassed, e. g., by sending a plain object that contains a pointer to shared resources. Instead, Rust leverages

the language compiler for checking that each value at any time has only one variable (called owner)[1].

Concerning the FastFlow parallel library, the point is that the potentially wrong usage of the reference-passing capability approach, which is at the base of the FastFlow programming model, is not checked by the library, and the potential faulty behavior is not signaled to the user. The programmer must properly use the provided mechanisms according to the FastFlow programming model.

Those three messaging approaches represent a spectrum of different policies. FastFlow focuses on full programmer control and high-performance thus, it does not guarantee any isolation property between concurrent entities. Instead, both CAF and Rust, even though at different levels, impose constraints to the user to provide statically checked guarantees. In the next sections, we will address the potential issues of the memory isolation property.

5.3 STATICALLY CHECKED ISOLATION IN DATAFLOW PROGRAMS

Enforcing isolation is particularly useful to the user to avoid common mistakes that could also lead to subtle data-races. The isolation enforcement through a statically checked capability system is also crucial to avoid potential run-time overheads. In the following, we show that it is possible to implement some FastFlow library's basic structures (i. e., pipeline and task-farm) using the features of the Rust programming language. We discussed the features of the Rust language in Section 2.4.2.

Indeed, Rust is a modern system-level programming language without a garbage collector whose code is compiled through the LLVM compiling infrastructure to native machine code. Rust focuses on enforcing memory safety, and it limits the usage of low-level tricks often used on C/C++ to achieve raw performance [217]. Nevertheless, we will demonstrate that, by using the Rust type system, it is possible to statically enforce the FastFlow library's message-passing semantics without dropping performance at run-time.

To have a fair performance comparison between implementations, we need to implement the FastFlow communication channel in Rust. Initially, we considered using the Multi-Producer Single-Consumer (MPSC) unbounded queue provided by the Rust standard library, but we discovered that it does not deliver the expected performance, in particular in the case of fine-grained computation. Therefore, we decided to port in Rust the C++-based FastFlow lock-free Single-Producer Single-Consumer (SPSC) unbounded queue [15].

Instead of writing it from scratch, mimicking the same FastFlow implementation, we decided to create a memory-safe Rust interface

1 Rust permits to lift the ownership rule inside the `unsafe block`, but this practice is used only in particular situations.

on top of the original C++-based FastFlow queue. The name of the Rust interface for the queue is `ff_buffer`[2].

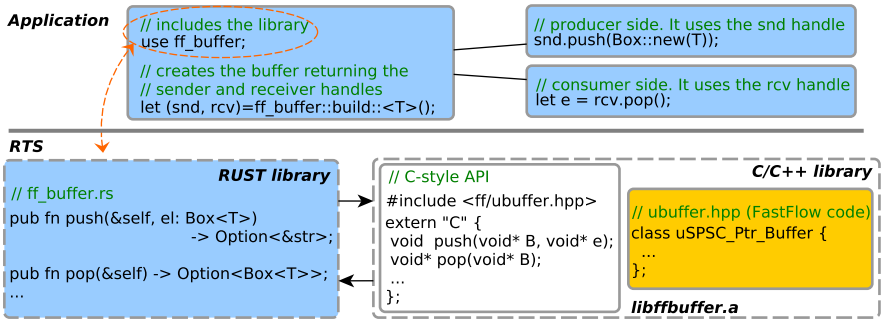


Figure 16: Integration of the FastFlow's unbounded SPSC lock-free queue in Rust.

Figure 16 shows the logical schema of the `ff_buffer` library that we used to integrate the FastFlow queue in Rust. The implementation is composed of two distinct parts: the Rust API providing a memory-safe interface of the queue, and the static C library that exposes the "unsafe" C interface of the C++ implementation. The `ff_buffer` library can be directly compiled as a standard Rust library. Moreover, it is possible to use the *Cross Language Linking Time Optimization*[3] feature of the LLVM compiler infrastructure to reduce the overhead of jumping back and forth between Rust and C++.

Another FastFlow feature we decided to use in the experiments is the ability to automatically pin all the spawned threads to distinct machine cores to improve the application performance when the number of threads is less than or equal to the available machine cores. For this purpose we used the Rust third-party library `core_affinity`[4] to set the thread-to-core affinity for all Rust threads according to a simple round-robin assignment strategy.

We considered two simple micro-benchmarks based on two FastFlow parallel patterns, namely the *Task-Farm* and the *Pipeline* (cf. Section 2.3.2). We selected these two patterns because they are used within the FastFlow library as basic building blocks to implement other more complex parallel patterns (e. g., D&C, Macro-Data Flow, ParallelFor).

In Figure 17 and Figure 18 we sketched the implementation schemes of the two FastFlow parallel patterns that we used as benchmarks for comparing the performance of the C++ and Rust versions. Figure 17 is the implementation of the *Task-Farm* pattern where the pool of Workers is composed of sequential nodes. Each node is implemented as a thread. Each Worker performs a configurable number of floating-point operations on each input data

2 Git repository link https://github.com/lucarin91/ff_buffer
3 http://blog.llvm.org/2019/09/closing-gap-cross-language-lto-between.html
4 https://crates.io/crates/core_affinity

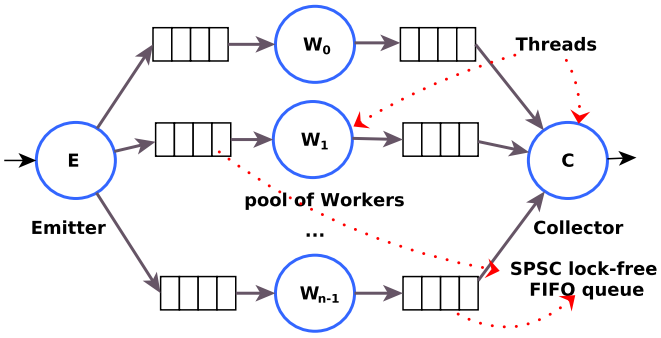


Figure 17: Implementation schema of the *Task-Farm* pattern.

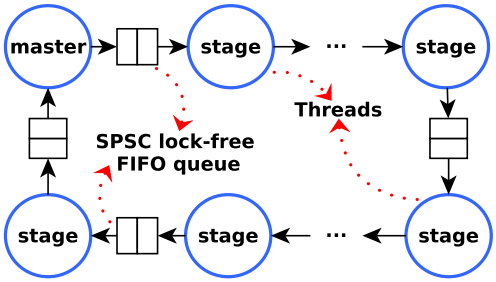


Figure 18: *Pipeline* with feedback channel.

element. The Emitter node is in charge of assigning data elements to the Workers according to a pre-defined or user-defined scheduling policy. We considered a simple round-robin assignment. The data elements produced by the Workers are all collected by the Collector node. This test aims at studying the scalability of the Task-Farm pattern by varying the number of Worker threads. Figure 18 shows the *Pipeline* with feedback pattern as implemented in FastFlow. In the tests we executed, we considered a Master stage (the first one) and a configurable set of other stages. The Master stage is in charge of generating a fixed-length stream of data elements in batches. The other stages of the *Pipeline* chain only forward the input element received to the next stage. The last stage of the *Pipeline* is connected to the Master stage, forming a circular *Pipeline*. This test aims to study the maximum throughput sustained by the *Pipeline* pattern by varying the number of stages.

The tests reported in this section were conducted on an Intel Xeon Server (full specification in Section 5.1) and repeated ten times. The values reported in the plots are the average value of all runs. The standard deviation is small (less than 1%) and thus omitted for readability reasons. We used the GNU `gcc` compiler version `7.2.0` with the `-O3` optimization flag enabled and the `rustc` compiler version `1.38.0` with `opt-level=3`.

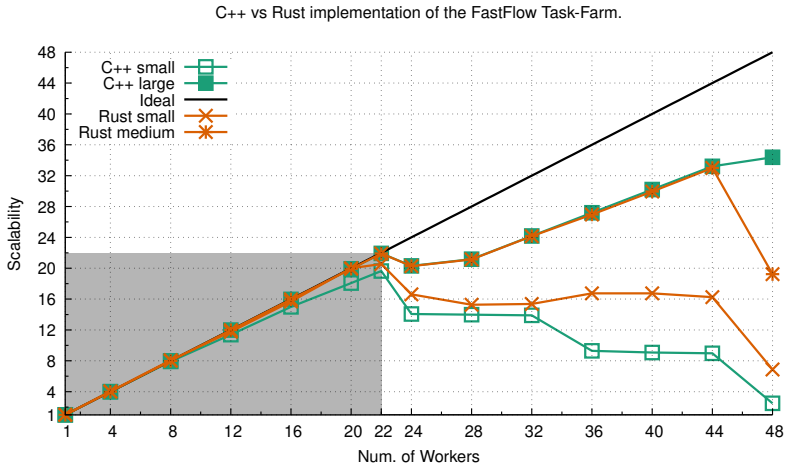


Figure 19: Scalability of the Task-Farm micro-benchmark implementation with two different computation granularities.

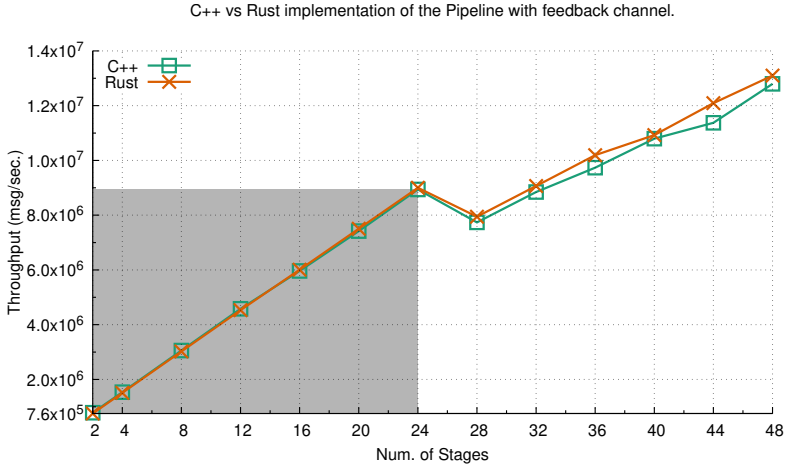


Figure 20: Sustained throughput of the *Pipeline* micro-benchmark with feedback channel varying the number of stages.

For the *Task-Farm* pattern we considered a stream of 50.000 elements and two different per-element computation granularities: *small* (about ~5 µs), and *large* (about ~5 ms). Figure 19 shows the scalability of the Task-Farm pattern written in C++ (i. e., FastFlow v3.0) and in Rust, respectively. The results show that the two versions have similar performance figures both for the *small* and *large* test cases. Both versions exhibit good scalability figures when the number of total threads used (that is equal to the number of Workers plus two) is less than or equal to the number of physical cores of the machine (this is the gray area of the plot). The Rust implementation of the benchmark uses a more simple (and aggressive) dequeueing strategy than the one offered by the FastFlow library. Moreover, the Rust version leverages the `jemalloc` memory allocator. These two optimizations allow to slightly improve the performance of the Rust version in the

small test case when the number of Workers is high. Conversely, for the *large* test case, the more aggressive polling approach used in the RUST implementation produce more overhead when the number of threads is greater than the available logical cores (i. e., the case of 48 Workers).

For the *Pipeline* test case, we consider a total number of 1M elements divided in an initial batch of 1K elements and 4K small batches each one containing 256 elements. Figure 20 shows the number of messages exchanged per second by varying the number of stages of the *Pipeline* chain. The performance of the two versions is almost the same, and the throughput increases almost linearly with the number of stages with a small drop corresponding to 24 *Pipeline* stages because from that point more threads than physical core are used.

The results obtained demonstrate no significant performance difference between the C++ and RUST versions when implementing data-flow patterns such as *Pipeline* and *Task-Farm* with undoubtful benefits in terms of programmability for the parallel programmer. However, this analysis is not enough because we also have to consider data-parallel computations in which more complex data isolation semantics is required.

5.4 ISOLATION IN DATA-PARALLEL COMPUTATIONS

As discussed in the previous section, maintaining the concurrent entities in complete isolation avoiding data sharing provides strong guarantees, and it could be enforced without introducing performance degradation at run-time, at least for concurrent data-flow networks built by composing *Pipeline* and *Task-Farm* patterns. Concurrent entities in data-flow networks can manage their internal state and exchange data with other entities moving the ownership from one entity to another. However, things are more intricate in other kinds of computations based on data partitioning, e. g., data-parallel computations.

In a nutshell, in data-parallel computations, an input collection is split into partitions assigned to multiple concurrent entities that work in parallel to produce an output collection. In those computations, usually modeled with the *Map* and *Map-Reduce* Parallel Patterns (cf. Section 2.3.2.5), the data races can be prevented by assigning disjointed partitions to concurrent entities and forcing each of them to modify only their local data (*owner computes rule*). However, in many cases, the partitions are not disjoined, and read-write sharing of the input collection is needed. In these cases, data races could be avoided by introducing extra data copies, but they impact the amount of memory used and the execution time due to the copy overhead. Our aim is to evaluate the performance impact of maintaining data isolation in data-parallel computation.

On shared-memory systems, basically, we have two possible communication protocols capable of enforcing data isolation. We call them *create-move* and *share-create*.

In the *create-move* protocol, the sender node (which is the master node of a *Map* pattern) creates the partitions by copying the content from the input collection into multiple separate sub-collections. Then, it sends them to the set of consumer entities (called Workers) to perform the computation. The Workers work in-place on their input sub-collection since it is not shared with any other entities. All results computed by the Workers are then sent back to the master node (or to a different collecting entity), which prepares the output collection by copying the partial results received.

In the *share-create* protocol, the master node sends an immutable reference of the input collection to all Workers. Each Worker determines its input partition and creates a new local partition for storing the results. All results are then collected as in the *create-move* case.

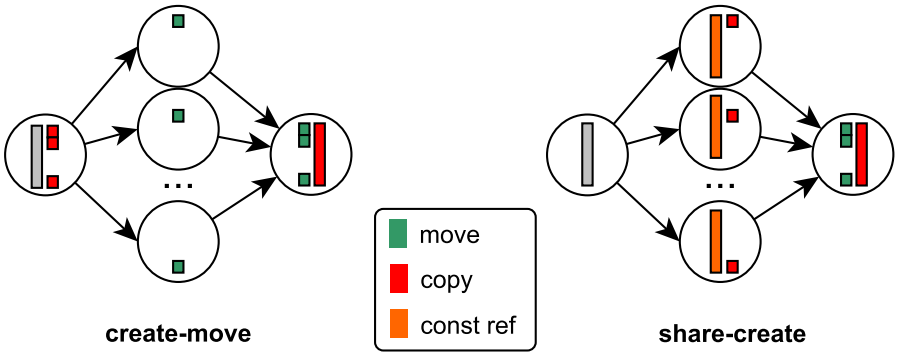


Figure 21: Two strategies for splitting and merging message data in data-parallel computations.

Figure 21 shows the two schemes for implementing the communication protocols needed in data-parallel computations to preserve the memory isolation property.

Although the two strategies are a general communication schema that can be implemented on many different programming languages and libraries, the *share-create* protocol requires the concept of *constant reference* to be safely implemented, i. e., `const &` in C++ or `&` in Rust. However, some widely used programming languages, e. g., Java and Python, do not limit the access of an object reference [58]. Therefore, in these cases, the only option is to implement the *create-move* strategy. However, both strategies can be implements in CAF, and in both cases, two copies of the entire input collections are needed. Moreover, the two strategies require to execute the same number of operations, but as we will show in the following, the *share-create* has some performance advantages.

The most optimized way to implement the protocol between the master node and the Workers is to share the original collection among

all entities allowing each Worker to modify *in-place* its logically assigned partition. However, this protocol does not provide any guarantee of data isolation. We will use it as a baseline reference to evaluate the overheads of the *create-move* and *share-create* protocols.

To evaluate the overhead of the two protocols, we built a simple data-parallel micro-benchmark in the C++ Actor Framework. CAF, as most Actors-based libraries/languages, forces the programmer to avoid data sharing among Actors preventing to send mutable references. As we already discussed in the previous section, CAF, as many others Actor Model libraries based on general-purpose languages, does not provide strong means to avoid data-sharing. Indeed, the CAF metaprogramming functionalities, which prevents the user from sending raw shared pointers between Actors, can be easily circumvent thought ad-hoc `struct` that encapsulates the raw pointer. In this framework, the idiomatic way to implement a *Map* computation working on an input collection is to use one of the strategies proposed above, thus introducing data copies.

We built two versions of the micro-benchmark, one implementing the `create-move` protocol and one implementing the `share-create` protocol. Then, we compare them with the baseline implementation (called `in-place`) implemented in C++ and using CAF low-level RTS features (i. e., we do not use directly Actors). The `create-move` version creates N partitions of an input vector and sends them to the N Worker Actors. The `share-create` version shares the input vector with the Worker Actors in read-only mode (i. e., by using a constant reference), and then the Workers internally allocate their local partition for storing the computed results. The Workers, do not execute any computation on the input partition. They just read and modify every single element of the assigned partition.

Table 4 shows the execution time of the three versions of the data-parallel micro-benchmark using two different size of the input vector: 12 MB, and 12 GB. The `share-create` version performs, on average, a bit better than the `create-move` version due to better utilization of the memory bandwidth. In fact, the `share-create` performs the allocation of the vector partitions and the data updates in parallel in all Workers, potentially exploiting all distinct memory channels available in the machines.

However, it is clear that both the `create-move` and `share-create` versions introduce significant overhead in both cases tested. These results, demonstrate that the impact on the performance of preserving the isolation property in data-parallel computations on shared-memory platform is a concrete issue. This result has important implications for the programming model on shared-memory platforms. On the one hand, the models that enforce isolation, such as the Actor Model, are particularly interesting from the programmability standpoint, helping the programmers avoid subtle data-races. On the other

Vector of 12 MB

	create-move	share-create	in-place
Xeon	13 ms	12 ms	3 ms
Epic	18 ms	12 ms	7 ms
Power 8	19 ms	18 ms	2 ms
KNL	51 ms	76 ms	12 ms

Vector of 12 GB

	create-move	share-create	in-place
Xeon	46.37 s	25.28 s	2.74 s
Epic	23.30 s	14.66 s	2.50 s
Power 8	13.61 s	11.52 s	2.18 s
KNL	29.16 s	17.33 s	4.10 s

Table 4: Execution time of the data-parallel benchmark implemented with the `create-move`, `share-create` and *in-place* protocols on different multi-/many-core platforms. The times reported correspond to the average value obtained by five distinct runs.

hand, they introduce excessive overheads in data-parallel computations, which represent a large fraction of applications, by preventing the direct exploitation of the platforms' physical shared-memory and thus limiting important optimizations.

5.5 SUMMARY AND DISCUSSION

In this chapter we have discussed the importance of the memory isolation property in parallel programming models targeting multi-/many-core platforms. The memory isolation property is an important property for guaranteeing memory-safety, but as we demonstrated with our experiments, it has an important impact on the performance of data-parallel applications on shared-memory platforms. Therefore, finding a balance between absolute performance figures and memory-safety is crucial for modern parallel programming models.

In Section 5.2 we introduced the isolation properties motivating the importance of having some forms of statically checked isolation in modern programming languages. To this end, we compared and contrasted different programming models providing different memory isolation guarantees, namely FastFlow, C++ Actor Framework (CAF), and Rust. In FastFlow, the user has complete control of the memory model. The library does not check any memory isolation property, leaving the responsibility of correctly using the model to the

programmers. CAF proposes a statically checked approach through C++ metaprogramming, which can be bypassed in particular circumstances by the programmer. Finally, the Rust language has a compiler that statically checks the so-called "ownership rule", i. e., data variables can have only one owner.

By using Rust, we showed that statically checked isolation does not introduce overheads in dataflow computations Section 5.3. However, its semantics strongly limits the possibility to reduce the memory copies in data-parallel computations, thus introducing overheads even when they could be avoided (Section 5.4).

We conclude by saying that to efficiently implement data-parallel computations on multi-/many-cores, memory isolation needs to be bypassed, but this has to be done without requiring direct intervention from the programmers and without discarding its benefits. As we will discuss in the next chapters, we propose a solution that raises the abstraction level by introducing well-defined software components whose implementation may break the memory isolation property. Still, they provide clean and memory-safe interfaces to the programmers.

6

PARALLEL PATTERN-BASED SOFTWARE ACCELERATOR FOR THE ACTOR MODEL

Parts of this chapter was been published in the Euromicro International Conference on Parallel, Distributed and Network-Based Processing *(PDP) [178].*

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. However, the high degree of freedom and flexibility offered by the Actor Model may easily lead to building complex Actor-based topologies that are hard to modify, debug, and most of all, optimize. Indeed, in such complex topologies, one or more Actors might become a bottleneck for specific types of messages or under some peculiar message rates. These situations are typically problematic to discover, given the intricated flow of messages. Once the programmers have identified the bottlenecks, they usually have to split the critical Actors into multiple distinct Actors and define a suitable communication protocol (e. g., master-worker, pipeline) among them to solve the bottlenecks. Accordingly, the application might need to be reconfigured to exploit the new Actors introduced.

To deal with these issues in Actor-based applications, we envision a synergy between the Actor Model and the structured parallel programming methodology based on parallel patterns. Critical parts of the application at hand can be modeled by using suitable parallel exploitation patterns (e. g., map, task-farm). Our first attempt to merge these two distinct parallel programming approaches is to use the concept of "Actors' accelerator". It is a software artifact that enables offloading parts of the computation to be accelerated without changing (or with minimal changes) the original Actor-based application structure. The accelerator is a separated entity of the application graph. It has its API to receive input messages from Actors and to send them the results.

This chapter describes our attempt to build the "Actors' accelerator" structure for data-parallel computations, specifically, *Map* and *Map-Reduce* parallel patterns. As shown in Chapter 5, data-parallel computations are challenging to implement efficiently on shared-memory systems using only messages in a model that enforces data isolation, like the Actor Model. Our implemented software accelerator for Actors runs on a partition of the platform's CPU resources and has been carefully designed to utterly integrate with the C++ Actor Framework (CAF) library.

Section 6.1 presents the design of the data-parallel accelerator describing its structure and how it interacts with the existing Actors. Then, Section 6.2 focuses on the CAF implementation of the accelera-

tor, which includes the extension of the CAF run-time itself including the *thread-to-core affinity* feature. We will show how the affinity feature enables the separation of the CAF run-time from the the data-parallel accelerator. Finally, Section 6.3 discusses the evaluation of the accelerator implementation considering two different benchmarks.

6.1 DESIGN A DATA-PARALLEL SOFTWARE ACCELERATOR

As described in Section 5.4 data-parallel computations are difficult to implement in an efficient way using the Actor Model, because of the strict requirement of maintaining data isolation among Actors. One of the objectives we have is to reduce message latency in data-parallel computations. In a nutshell, the scenario is that of an Actor receiving in input a relatively large data structure that has to be processed and sent to another Actor.

In the Actor Model, there is no idiomatic approach to split the input data, compute each part concurrently, and merge the single results in a new message ready to be sent out. This is a typical *Map*-based parallel computation structure, which could be provided to the Actor Model's programmer as a ready-to-use tool to solve these kinds of parallel computations. Indeed, several data-parallel patterns that offer clear functional and parallel semantics can be used to simplify parallel programming. They relieve the application programmers from designing and implementing well-known parallel structures, allowing them to concentrate on the application's business logic.

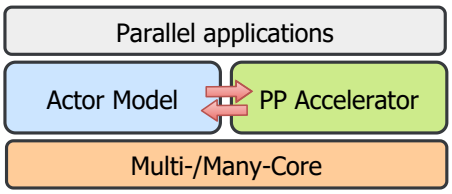


Figure 22: The layered software design showing the relations between the Actor Model and Actors' accelerator that leverages Parallel Patterns.

Figure 22 sketches our idea leveraging a software accelerator to provide a set of data-parallel patterns to Actors programmers. The accelerator coexists with the application Actors used to implement the application. Thus, the programmer may take advantage of Actors' flexibility without needing to reconfigure their application to speeding up data-parallel computations.

The *Map* pattern (cf. Section 2.3.2.5) is a data-parallel paradigm that applies the same function to every element of an 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.

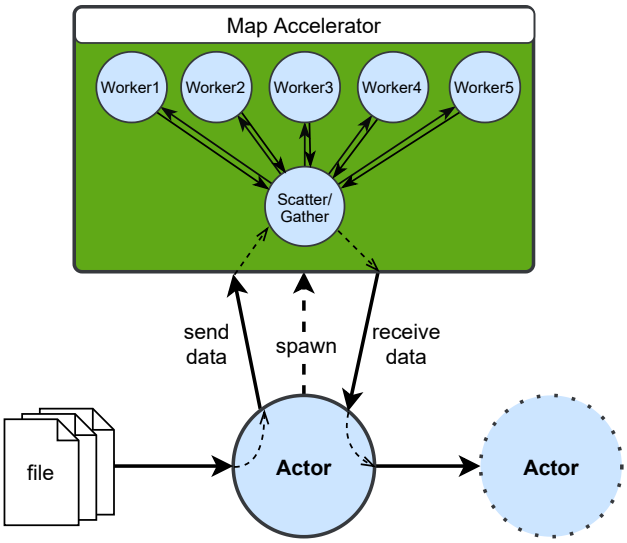


Figure 23: The logical schema of the the *Map* accelerator.

We designed the *Map* accelerator as a set of Actor 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 ready. 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. Figure 23 shows the design of the *Map* accelerator.

The internal is designed to follow the *share-create* communication schema, which we already described in the previous chapter (cf. Chapter 5). The accelerator maintains the isolation property of the Actor Worker used in the design of the *Map* pattern at the cost of introducing 2 extra copies of the computed structure. We will focus on eliminating this extra overhead in the next chapter. Here we focus on designing the software accelerator in a way that is easy to use for the Actor programmer and allows to increase the performance of data-parallel computations that otherwise would be executed sequentially by the single Actor. In Chapter 7 we propose a new solution where the *Map* pattern is implemented as a "macro Actor", whose implementation skeleton leverages shared-memory to reduce costly memory copies and in general to reduce communication overhead.

The design of the pattern's API and its clear parallel semantics aids the programmer avoid introducing data-races.

We implemented the Actor's accelerator in C++ by using a state-of-the-art Actor Model implementation: C++ Actor Framework (CAF) (cf. Section 2.4.1). Therefore, both the application Actors, as well as the accelerator, have been implemented in CAF. The accelerator can be implemented in other frameworks, but we decided to use C++ Actor Framework to reduce the possible overheads given by the interaction of different run-time systems. Indeed, C++ Actor Framework permits to spawn *detached* Actors, which are Actors that have their dedicated thread of control. We use detached Actors to implement the internals of the accelerator and use higher communication between threads. Moreover, to minimize the interferences introduced by the software accelerator structure, we enhanced the run-time of C++ Actor Framework to control the placement of system threads on different machine cores. In the following, we call this new C++ Actor Framework feature *thread-to-core affinity*. Using the *thread-to-core affinity*, we are able to confine the threads used to implement the accelerator to a restricted set of machine cores.

6.2 IMPLEMENTATION OF THE ACTORS' ACCELERATOR IN CAF

The implementation of the accelerator has been done at two different levels. 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 us to control the mapping of different CAF threads used to execute Actors, for example to confine the threads used for implementing the accelerator on a subset of the machine cores. At the top level, we designed and implemented the software accelerator and its API by using C++ Actor Framework Actors. Specifically, we implemented the *Map* pattern.

6.2.1 *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 from moving 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 set of system configuration parameters (i. e., `affinity-runtime`, `affinity-detached`, `affinity-blocking`), which allows to statically specify on which cores the different kinds of CAF 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 respects the following grammar (COREID is a valid core identifier):

```
affinitystr ::= grouplist
  grouplist ::= group | group grouplist
      group ::= < rangelist >
  rangelist ::= range | range, rangelist
      range ::= COREID | COREID – COREID
```

The `affinitystr` is composed by a set of groups enclosed in angle brackets (`< >`). A group hosts a collection of cores separated by commas (`,`) or a range of them delimited with a single dash (`-`). Threads assigned to a core group are forced by the OS to run only on the cores of that group. For instance, the group `"<0,2>"` specifies that all the threads assigned to that group are allowed to run on either core 0 or core 2. The OS is anyway free to suspend or move at each point in time each threads within the core group. The CAF run-time has been modified in such a way to read and parse the *affinity string* for each different kind of CAF threads, and execute the proper system calls for setting the thread affinity. Each new threads created by the CAF library is placed on the next core group of the list. When the list ends it restarts from the first one. 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.

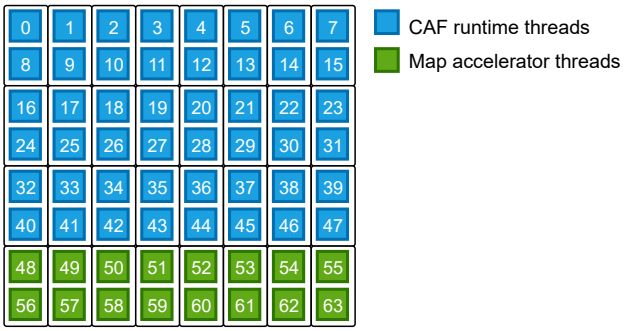


Figure 24: Example of *thread-to-core affinity* on a 64 CPU cores Intel KNL platform. Configuration used: `affinity.scheduled-actors=<0-47>` `affinity.detached-actors=<48-63>`.

Figure 24 shows a simple example where the 64 CPU physical cores of an *Intel KNL* platform (cf.. Section 5.1) has been partitioned be-

tween the CAF run-time threads and the threads used for running the Actors implementing the *Map* accelerator. This partitioning could be achieved by supplying the `"<0-47>"` `affinitystr` for the CAF run-time threads and `"<48-63>"` for the CAF detached Actor threads used for the accelerator.

6.2.2 *Map pattern 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 Figure 23). 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 provided in the request message 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, waits for an incoming chunk of data elements and the function to apply to each item of the collection. All Workers read from the same data collection, which is shared as read-only reference (i. e., `const &`) 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.

```
1  /* Spawn a map accelerator instance */
2  auto map_instance = caf::spawn(map, 5);
3  /* Declare the Map function */
4  auto F = [](int el){ return el + 1;};
5  for (auto&& vec : read_from_disk()){
6     /* Offload the computation to the accelerator */
7     caf::request(map_instance, F, std::move(vec)).then(
8        /* Async receive */
9        [=] (std::vector<int>& result) {
10          send(next_actor, std::move(result)); // send the result
11    });
12 }
```

Listing 3: Simple example showing how to use the Actors’ accelerator to compute a *Map* Parallel Pattern.

Listing 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 Figure 23. The Actor creates a new *Map* accelerator instance and starts to offload data read from a file. Then the Actor asynchronously sends the results obtained from the accelerator to another Actor (*next actor* in the figure).

In particular, in 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.

In order 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.

6.3 EVALUATION

In this Section, we test the CAF implementation of the *Map* accelerator discussing both a simple synthetic benchmark and also a modified version of the CAF Latency Benchmark described in [211].

All tests were executed on a *Intel Xeon Phi*, code-name *Knights Landing*, KNL. The KNL is equipped with 64 physical cores and 256 logical one. The complete specification of the machine can be found in Section 5.1. The code was compiled with GNU `gcc` compiler version 7.3.0 and 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 accelerator executing 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 µs 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.

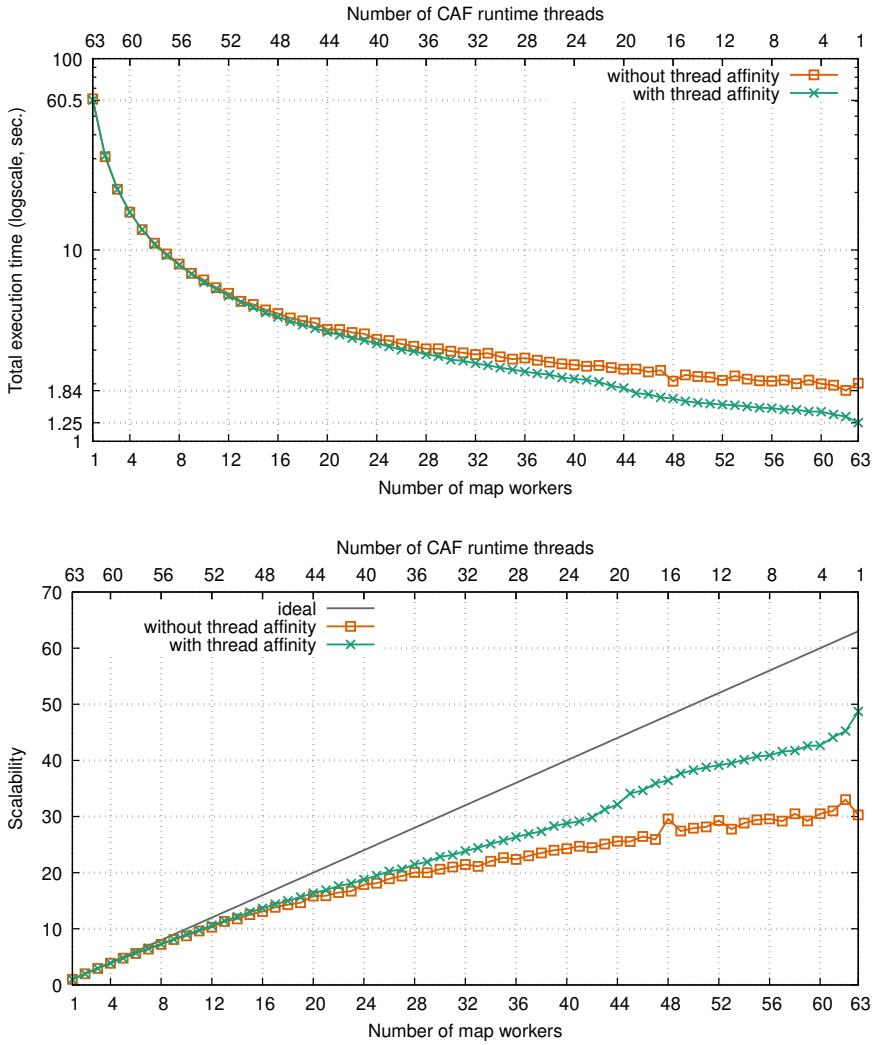


Figure 25: The execution time (top) and the scalability (bottom) of the data-parallel benchmark on the KNL platform.

Figure 25 shows the execution time (in seconds) and the scalability of the test, respectively. 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 observed, 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.

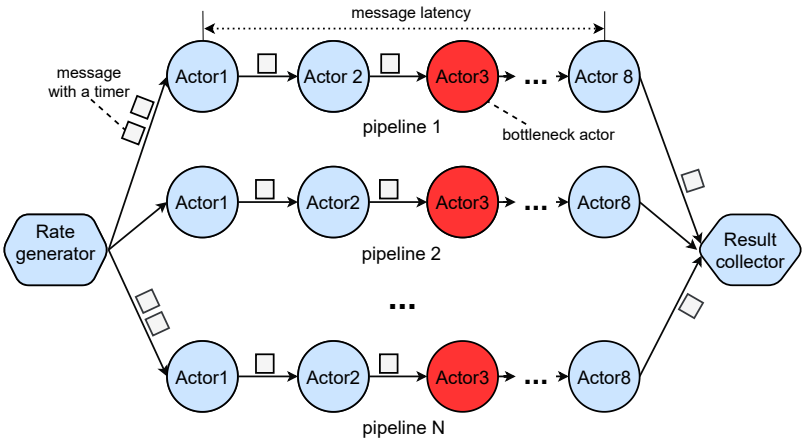


Figure 26: The CAF Latency benchmark with bottleneck Actors.

As a second test, we considered the CAF Latency Benchmark [211]. It aims at measuring the message latency of CAF's Actors considering either a single pipeline of Actors or multiple replicas of the pipeline chains each one having a fixed number of Actors (see Figure 26). A *Rate generator* Actor generates messages at a given constant rate. The *Result collector* Actor, collects all messages and computes the average message latency.

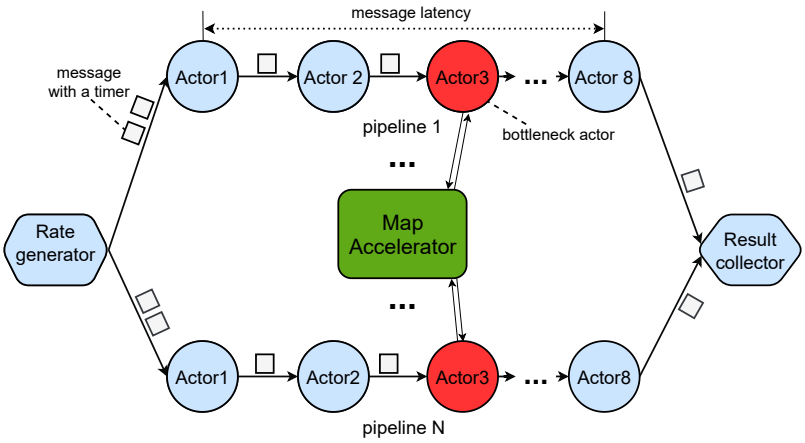


Figure 27: The *Map* accelerator in the CAF Latency benchmark.

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 Figure 27). 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 1000 messages per second. The computing Actors work on an input collection of 5000 elements. The average execution time per item is about 1 µs. The benchmark lasts 15 seconds.

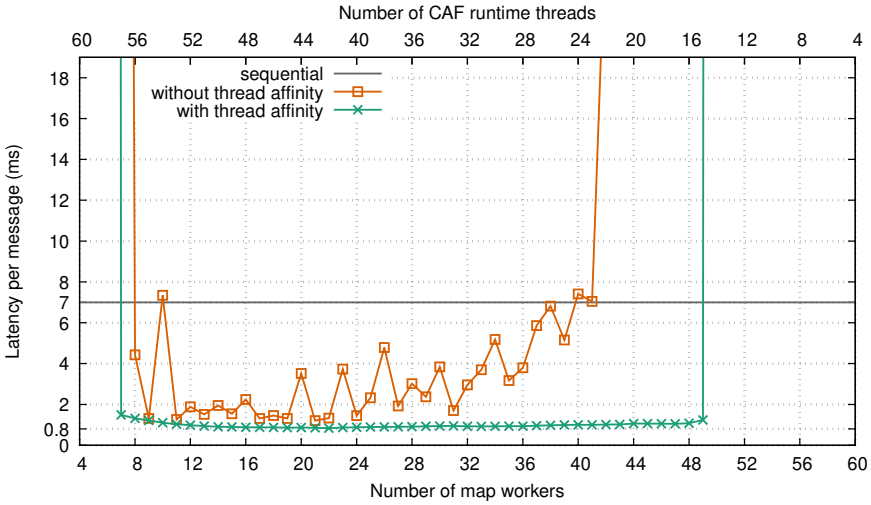


Figure 28: CAF Latency Benchmark with 100 chains and input vectors of 5000 elements.

Figure 28 shows the average message latency of a message for traversing one 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 total number of Actors implementing the benchmark (i. e., 800), respectively. Therefore, in such conditions, 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 7 ms, whereas the version with the *Map* accelerator and

the *thread-to-core affinity* enabled has an average latency of 0.8 ms. 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 Actors' accelerator implemented can reduce the service time of Actors performing map-like computations (i. e., pure map and map+reduce), requiring only a minimal programming effort to the application developer for defining and spawning the accelerator.

6.4 SUMMARY AND DISCUSSION

This chapter focuses on providing ready-to-use Actors' accelerator running predefined Parallel Patterns. We concentrated on data-parallel patterns such as *Map* and *Map-Reduce*. The accelerator has been built with the aim of interacting with application Actors seamlessly and with minimal modification to the application structure. It enables the speedup of data-parallel computations, otherwise implemented sequentially within a single Actor. The software accelerator is spawned "beside" the Actors implementing the application, with separated run-time threads. Multiple distinct Actors can interact with it by offloading data-parallel computations and waiting for the results.

In Section 6.1 and Section 6.2 we addressed the design and the implementation on top of the C++ ACTOR FRAMEWORK (CAF). To avoid potential conflicts between the distinct run-time threads used to execute the Actors and those used for the software accelerator, we designed and implemented an extension of the CAF framework to control, at fine granularity, the threads' placements into core resources (thread-to-core affinity). Finally, in Section 6.3 we showed that the *Map* accelerator is capable of speeding up Actor-based computations removing potential bottlenecks.

The accelerator concept can also be further extended to support other data-parallel computations, e. g., *stencil*-based computations. Nevertheless, we decided to follow a different path to synergically combine the Parallel Pattern-based approach with the Actor Model in a single monolithic programming model. In the next chapter, we propose this new approach based on the tight integration of Parallel Patterns and Actors, where Parallel Patterns behave as "macro Actors". Indeed, macro Actors and Parallel Patterns are used to structure the application and communicate with standard messages sharing the same RTS.

7

EFFICIENT PARALLEL PATTERNS FOR THE ACTOR MODEL

Parts of this chapter was been published on the International Journal of Parallel Programming *(IJPP) [176]*

The chapter describes our proposal integrating the Parallel Patterns parallel programming approach into the Actor Model for multi-/many-core target platforms. We focus on using Parallel Patterns to bring into the Actor Model both enhanced programmability leveraging well-known parallel components and performance optimizations in the execution of data-parallel computations. Parallel Patterns are integrated into the Actor Model as "macro Actors". In other words, in the Actors' world, they are seen as standard Actors and can communicate with other Actors through regular messages. Instead, the implementation skeleton of a given Parallel Pattern exploits multiple concurrent entities and shared-memory communication mechanisms that are totally transparent to the outside Actors' world.

We have already discussed in Section 5.4 how the isolation properties of the Actor Model can introduce unnecessary overheads in data-parallel computations. Using our set of data-parallel patterns, i. e., *Map* and *Divide&Conquer*, which introduce low-level shared-memory-based optimizations otherwise precluded by the "pure" Actor Model, the user may benefit from extra performances without renouncing to the Actor Model's guarantees, such as memory isolation.

We start this chapter by describing the adopted design principles that we followed to build our set of Parallel Patterns (i. e., Section 7.1). Specially, we focused on the need to have a memory-safe integration of Parallel Patterns into the Actor Model while exploiting low-level optimization features of the hardware platforms. Then, Section 7.2 and Section 7.3 present the implementations of the Parallel Patterns by using the CAF library. In particular, Section 7.2 presents the *Data-parallel Patterns* implementation, whereas Section 7.3 presents patterns suitable to be used to allow combination and nesting of Parallel Patterns to build a structured composition of Actors, i. e. *Control-parallel Patterns*. Finally, in Section 7.4, we evaluate our approach by implementing a set of applications selected from the Parsec benchmark and the Savina benchmark suites, comparing the performance of our approach with those obtained by other specialized parallel libraries targeting multi-/many-core platforms.

7.1 DESIGNING PARALLEL PATTERNS AS ACTORS

By employing Parallel Pattern abstractions, we aim at bringing to the Actor Model two crucial aspects when considering modern multi-

/many-core platforms: a) to introduce a software layer in which we can apply low-level optimizations without breaking the isolation principle of Actors; b) to provide ready-to-use and well-known communication structures in Actor-based parallel applications.

Introducing low-level optimization leveraging on physical shared-memory, is generally not allowed by the "pure" Actor Model without breaking the principles of the model itself. During the design of Parallel Patterns, we used the following guidelines:

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

We designed the Parallel Patterns 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. Each Parallel Pattern can be seen as a "macro Actor". It means that, from the Actors programmer standpoint, a single Parallel Pattern or a combination of Parallel Patterns behaves like an Actor. Conversely, their implementation skeleton is composed of multiple concurrent entities cooperating to implement the pattern's parallel semantics.

We decided to implement these Parallel Patterns 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 Parallel Patterns 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*, cf. Section 2.4.1). 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.

We divided the Parallel Patterns in two sets[1]:

- *Data-parallel Patterns* namely *Map*, *Divide&Conquer*; and
- *Control-parallel Patterns* namely *SeqActor*, *Pipeline*, and *Farm*.

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

The first set of patterns internally exploits shared-memory to optimize the performance. The second set, enables nesting and composition of Parallel Patterns focusing more on structuring the concurrent graph of Actors and Parallel Patterns.

7.2 DATA-PARALLEL PATTERNS

As described in Section 2.2, data-parallel computation focuses on partitioning the input data and applying some transformation in parallel on each partition. Those kinds of computations profit from shared-memory optimizations, which are not allowed in the "pure" Actor Model (cf. Section 5.4). In the following, we discuss our implementation as "macro Actors" of the *Map* and *Divide&Conquer* Parallel Patterns.

7.2.1 *Map*

The *Map* pattern is a data-parallel paradigm that applies the same function to every element of an input collection (see Section 2.3.2.5). 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.

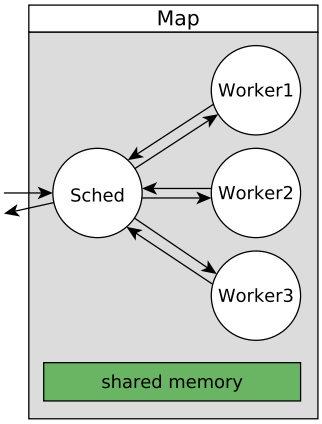
The efficiency of the Map pattern on multi-cores depends on the ability to share the input collection on which the user function has to be applied. Data-races are avoided by design because the parallel semantics of the Map pattern is such that distinct concurrent entities work on disjoint sub-collections.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

The pattern API enforces correctness. Indeed, the user will only supply a C++ lambda that applies the computation only on the provided partition of the input container, and in no ways, it will not be able to access the other partitions.

Figure 29 (left-hand side) 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.

Figure 29 (right-hand side) shows how to configure and spawn a *Map* pattern with the Parallel Pattern 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.

Possible configuration are the number of internal Workers (i. e., `replicas`), the type of scheduling policy (i. e., `scheduler`), and the used run-time (i. e., `runtime`). The `replicas` configuration parameter, if not set, is equal to the number of active cores (Line 6). The `scheduler` parameter can be set to either `static` assignment of partitions (the default value) or `dynamic` assignment of partitions (Line 7). Finally, the `runtime` parameter can be set to `actors`, which uses *scheduled* Actors coordinated by the work-stealing run-time of CAF, or `threads`, which uses *detached* Actors that have their own thread of control (Line 9).




```
1  using Cnt = std::vector<int64_t>;
2  auto map = Map<Cnt>([](auto range) {
3     for (int64_t &el : range) {
4        // do somethings with el
5     }
6  }).replicas(3)
7    .scheduler(PartitionSched::static_{})
8  //.scheduler(PartitionSched::dynamic_{1})
9    .runtime(Runtime::actors);
10 //.runtime(Runtime::threads);
11 auto p = spawn_pattern(sys, map).value();
```

Figure 29: On the left-hand side the *Map* pattern implementation scheme, and on the right-hand side the code to build and spawn the *Map* pattern. The lines commented, show different options for the scheduling policy and the kind or RTS used, respectively.

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 remarking 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 directly defined by the application programmer.

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.

In the evaluation of Section 7.4, we will show that using the `threads` run-time it is possible to improve performance avoiding the indirection and overheads of adopting the work-stealing scheduler for the ready Actors. Nevertheless, using a number of threads higher than the number of the available CPU cores could lead to CPU resource overprovisioning, with extra overheads due to the CPU cores' contention. This is particularly evident when the computation of Actors is mainly CPU-bound. For these reasons, we decided to maintain both options and to permit the user to select between the two run-times depending on the specific situations.

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* Parallel Pattern 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 reverse order and produces the output result. This implementation guarantees that the shared-memory layer inside the Parallel Pattern is transparent to the application programmer.

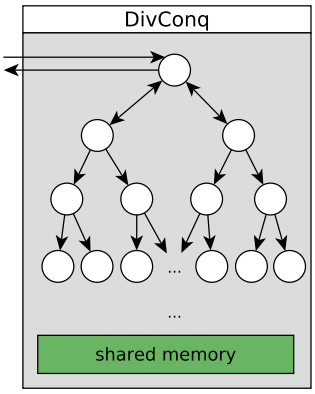
7.2.2 *Divide and Conquer*

The *Divide&Conquer* (*D&C*) programming approach is used in several well-known algorithms, such as *MergeSort*, and the *Strassen algorithm* for matrix multiplication. In *D&C* algorithms, during the Divide (or Split) phase, the problem is recursively decomposed into smaller sub-problems building a tree of function 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 partial results that eventually are combined to produce 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.

Our implementation of the *D&C* Parallel Pattern follows the design of Mattson, Sanders, and Massingill [148] described in Section 2.3.2.7. Concretely, the design is based on the definition of three custom functions (i. e., *divide*, *merge* and *sequential*) and a boolean condition that decide to stop the recursion.




```
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();
```

Figure 30: On the left-hand side the *DivConq* pattern implementation scheme, and on the right-hand side the code to build and spawn the *DivConq* pattern.

As shown in Figure 30 (left-hand side), 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*.

Figure 30 (right-hand side) shows the interface of the *DivConq* Parallel Pattern. The pattern is created by passing a user defined *Container* and four functions that work on continuous portions of the input container called *Range*s (`Rng` in Figure 30). Ranges are essential software components in our data-parallel patterns implementation. They permit to split a data container into disjointed partitions, avoid-

ing costly copying operations and maintaining the isolation property at the same time. Indeed, ranges are only continuous references to a subset of data elements of a container. We uses the `range-v3`[2] library to provide the ranges support to our data-parallel patterns. `range-v3` is an abstraction layer on top of standard iterators. It supports all standard C++ containers and can be used in combination with all containers exposing standard C++ iterators. Important features of ranges are the *Views* and the *Actions*. *Views* are composable adaptations of ranges where the adaptation happens lazily as the view is iterated. Instead, *Actions* is an eager application of an algorithm to a container that mutates the container in-place and returns it for further processing. The combination of those transformations permits the complete manipulation of ranges inside the user-code of the data-parallel patterns.

7.3 CONTROL-PARALLEL PATTERNS

The *Control-parallel Patterns* are used particularly to combine and nest multiple Parallel Patterns to build complex computational graphs. In our discussion, we consider the *Sequential*, *Farm*, and *Pipeline* control-parallel patterns.

The primary objective of the Control-parallel Parallel Patterns is to provide well-known parallel structure and to enhance programmability. However, they also enable important optimizations, specifically for streaming computations, as we will examine in depth in Chapter 8. Indeed, in Chapter 8 we will describe a new optimized version of the following patterns for the high-throughput streaming application domain.

7.3.1 *Sequential*

The *Sequential* pattern represents a single concurrent entity part of the *Pipeline* and *Farm* patterns. We developed two types of *Sequential* components: *SeqActor* and *SeqNode*. Both *SeqActor* and *SeqNode* were designed to integrate the user code inside the Parallel Patterns. They are not meant to be spawned alone, but to be used inside *Control-parallel Patterns*.

The *SeqActor* can be used to encapsulate a user-defined Actor within Parallel Patterns. The *SeqNode*, is instead a custom sequential component capable of optimizing the exchange of messages between two distinct *SeqNode*(s). It provides specific optimization for high-throughput streaming applications modeled as a composition of *Farm* and *Pipeline* patterns.

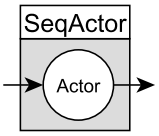
Although the *SeqActor* does not introduce specific optimizations for the message exchange between Actors, it enables using an already

2 https://github.com/ericniebler/range-v3

defined Actor within Parallel Patterns directly. Therefore, it has been designed mainly to facilitate the integration of Parallel Patterns in the Actor Model. Indeed, using *SeqActor*(s) in combination with the *Pipeline* and *Farm* patterns allow us to introduce well-defined communication structures to an already defined Actor-based application.

In the following we examine the *SeqActor*, instead, the *SeqNode* details will be provided in Chapter 8.

The *SeqActor* pattern can be seen as a factory of a specific CAF Actor. It allows to spawn different copies of the same Actor and use them in different points of a Parallel Pattern composition. *SeqActor*, other than the standard Actor functionalities, also provides mechanisms to enable Actors' composition. It maintains the references to the next Actors in the *Pipeline* chain. Such references are automatically set when the *SeqActor* is used within *Control-parallel Patterns*. The programmer can use the `send_next` method to forward messages to the next Actor in the *Pipeline* chain. This function enables a straightforward composition of Actors, allowing the programmer to reuse the same Actor definition within different Parallel Patterns.



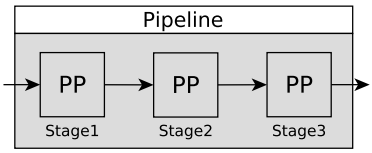

```
1  struct MyAct : public pp_actor {
2   MyAct(caf::actor_config &c, optional<Next> n)
3          : pp_actor(c, n) { }
4   caf::behavior make_behavior() override {
5     return /* my behavior definition */;
6   }
7  };
8  /* ... */
9  // with no initialization
10 SeqActor<MyAct> seq1;
11 // with initialization
12 SeqActor<MyAct> seq2{[&](caf::actor a) {
13  caf::anon_send(a, par1, par2);
14 }};
```

Figure 31: The *SeqActor* pattern implementation scheme (left-hand side). The example code for building a *SeqActor* by using the `MyAct` CAF Actor (right-hand side).

The left-hand side of Figure 31 shows the implementation scheme of the *SeqActor* pattern. Instead, the right-hand side of Figure 31 shows the definition of a new Actor, called `MyAct` and two different ways of creating a *SeqActor* pattern from `MyAct`. In the first case (Line 10) the Actor will be initialized without any parameter. In the second case (Line 12), the programmer provides a callback that will be called as soon as the Actor will be spawned to initialize it.

7.3.2 *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.



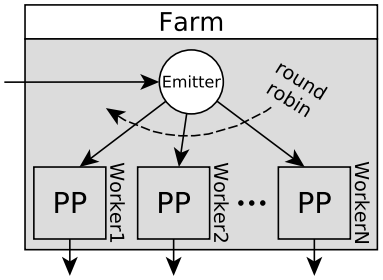

```
1 SeqActor<MyAct1> s1;
2 Map<Cnt> s2{[](auto range){/*...*/}};
3 SeqActor<MyAct2> s3;
4 Pipeline pipe{s1, s2, s3};
5 auto p = spawn_pattern(sys, pipe)
6          .value();
```

Figure 32: In the left-hand side the *Pipeline* pattern implementation schema. An example code for building and spawning an instance of a three-stage pipeline on the right-hand side.

The *Pipeline* pattern stages can be any nested sequence of Parallel Patterns presented in this section. Figure 32 shows the schema of the *Pipeline* pattern and a simple example of how it can be built and spawned as any other Actor. The *Pipeline* pattern takes care to connect each stage in the correct order, according to the precedence order sets by the user in its constructor (Line 4 of Figure 32).

7.3.3 *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. Precisely, a the *Farm* pattern replicates a given number of times the Parallel Pattern provided as Worker. As for the *Pipeline*, the generic Worker can be any valid composition of the patterns presented in this section. The number of Worker replicas can be left unspecified, meaning that a default value will be used (e. g., the number of active CPU cores). Finally, the *Farm* implementation supports three scheduling policies, namely `round-robin`, `broadcast`, and `bykey`. The `round-robin` policy forwards the incoming messages to different Workers in a round-robin fashion, i. e., restarting from the first one once all Workers received a message. Instead, the `broadcast` policy forwards the same incoming message to all Workers. This policy exploits the CAF *copy-on-write* feature, thus the message is automatically copied only if the receiving Workers update the input message. Lastly, the `bykey` policy uses a configurable field of the incoming message as a key-value, and it forwards all the messages with the same key-value to the same Worker. The `bykey` policy is extremely common in streaming applications where the *Farm* Workers manage a partition of a distributed state (cf. Section 8.3).




```
1 SeqActor<Worker> worker;
2 auto farm = Farm{worker}
3             .replicas(N)
4             .policy(round_robin());
5 auto p = spawn_pattern(sys, farm)
6          .value();
```

Figure 33: The *Farm* pattern implementation schema (left). An example code for building a *Farm* pattern with N sequential Workers and the *round-robin* policy (right).

Figure 33 shows both the internal implementation scheme of the *Farm* pattern and a simple example code for instantiating it.

7.3.4 *Composition of Parallel Patterns*

The patterns in the Parallel Patterns library, can be composed to build more complex computation structures according to the "two tier model" (see Section 2.3.2), which is formalized in the following grammar.

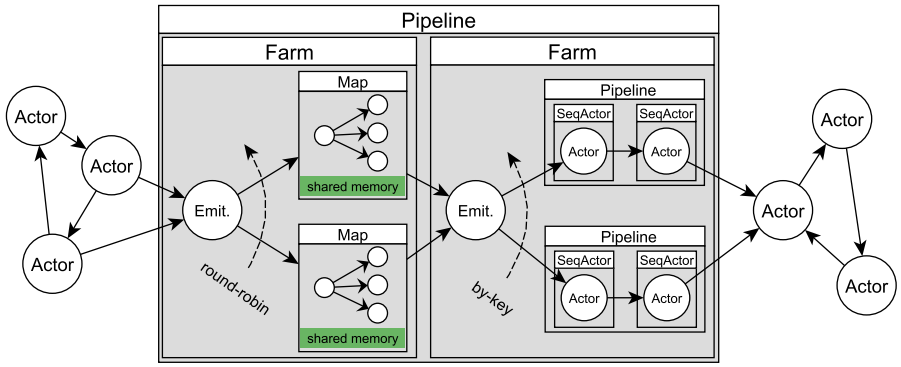
$$\textbf{PP} := \textbf{Node} \mid \text{Farm}(\textbf{PP}) \mid \text{Pipe}(\textbf{PP}+) \tag{1}$$

$$\textbf{Node} := \textit{SeqActor} \mid \textit{SeqNode} \mid \textit{Map} \mid \textit{D\&C} \tag{2}$$

The *Farm* and *Pipeline* patterns can have as internal elements any patterns, while *SeqActor*, *SeqNode*[3], *Map*, and *Divide&Conquer* must be the leaf-nodes of the skeleton tree composition representing the application (or part of it). Specifically, the *Map* and *Divide&Conquer* cannot contain other Parallel Patterns.

Figure 34 (top side) 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. Moreover, the code in the bottom side of Figure 34 shows how to define and instantiate the Parallel Patterns composition sketched in the above schema. In particular, Line 1 to Line 6 define the first *Farm* composed of two *Map* patterns. Line 8 to Line 13 define two *SeqActor*s connected in pipeline and create a second *Farm* with the composed *Pipeline*. Finally, Line 15 and Line 16 build the external *Pipeline* composing the two already defined farms and spawn it. The standard Actors of the application may feed the Parallel Pattern by

3 The complete specification will be provide in Chapter 8.




```
1  auto map = Map<std::vector<int64_t>>{/*...*/}
2            .scheduler(PartitionSched::static_{})
3            .replicas(3);
4  auto farm1 = Farm{map}
5              .policy(user_defined)
6              .replicas(2);
7
8  SeqActor<act1> seq1{/*...*/};
9  SeqActor<act2> seq2{/*...*/};
10 Pipeline pipe{seq1, seq2};
11 auto farm2 = Farm{pipe}
12             .policy(round_robin{})
13             .replicas(2);
14
15 Pipeline pattern{farm1, farm2};
16 auto app = spawn_pattern(sys, pattern).value();
```

Figure 34: Composition of two *Farm*s using a *Pipeline* pattern. The composition schema (top) and the code to build the pattern composition (bottom). The first *Farm* has a *Map* pattern as Worker replicated two times, whereas the second *Farm* uses *Pipeline* of *SeqActor* as Workers.

using the `app` handle provided by the spawn operation in Line 16 in the same way they use the handles of other standard Actors.

7.4 EVALUATION

In this section, we first highlight the performance problems of using the "pure" Actor Model on multi-core platforms and how the Parallel Patterns 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 Parallel Patterns library with that obtained by using the native PTHREADS implementation shipped with PARSEC, and the FASTFLOW implementation that uses the same Parallel Patterns approach to parallelize the benchmarks. The FASTFLOW performance of PARSEC benchmarks has been already compared with other specialized frameworks on multi-/many-cores

demonstrating comparable (and in some cases better) overall performance [85].

The experiments were conducted on two different multi-cores, i. e., `Xeon` and `Power8` (cf. Section 5.1). All experiments have been executed several times and the average value obtained has been used to compute the speedup shown in the following plots. The standard deviation is generally low and not reported in the plots.

We consider a subset of SAVINA [126], and PARSEC [34] benchmarks:

- SAVINA is a set of benchmarks specifically conceived for evaluating Actor Model 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 require more computation (e. g., Matrix Multiplication). We selected two applications of the *parallelism benchmarks* set, namely `quicksort` and `recMM`, 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 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 [34, 53]. For testing the Parallel Patterns library, we selected `ferret`, `blackscholes`, `raytrace` and `canneal` benchmarks. 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.

7.4.1 *"Pure" Actor Model vs Actors+Parallel Patterns*

Here we compare the performance of the "pure" Actor Model with the Actor Model enriched with the Parallel Patterns 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" Actor Model semantics. In its implementation, there are not shared variables among Actors. During the Split and the Merge phases of

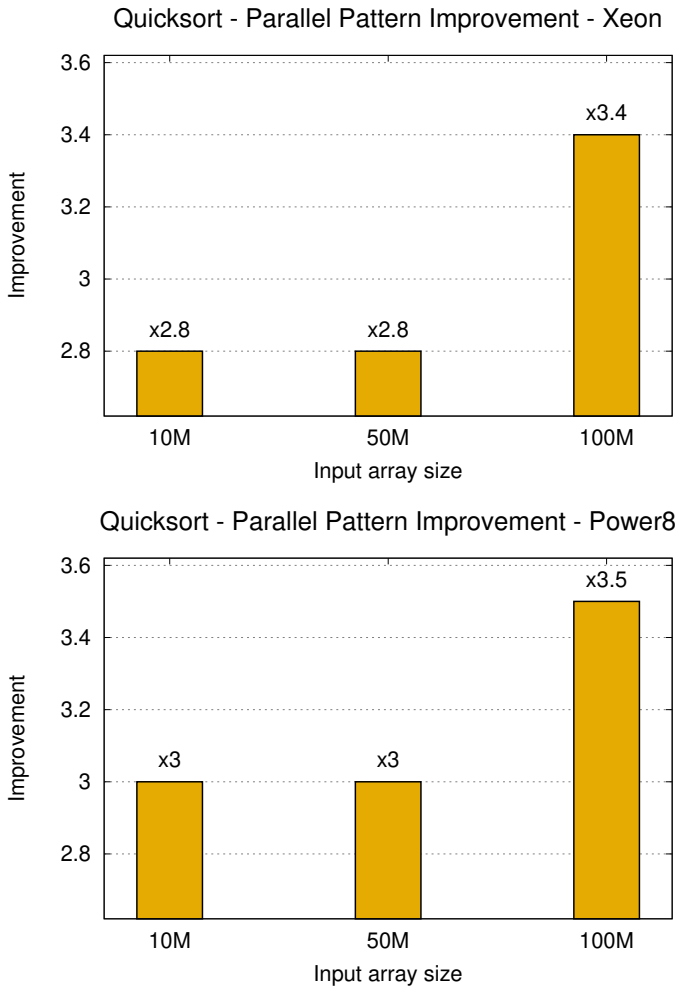


Figure 35: Improvement of the Parallel Patterns version compared to the "pure" Actor Model version of the `quicksort` benchmark on the `Xeon` and `Power8` platforms.

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 35 shows the performance improvement of the Parallel Patterns approach with respect to the "pure" Actor Model implementation of the `quicksort`, considering different sizes of the input vector, i. e., 10M, 50M, and 100M elements, and a fixed cut-off value of 2000 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 ~3.5) on the `Xeon` and ~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 Parallel Patterns version

vs the "pure" Actor Model implementation, respectively). The low scalability results are because the `quicksort` algorithm needs an initial phase of consecutive splits operations before became efficient on multiple cores.

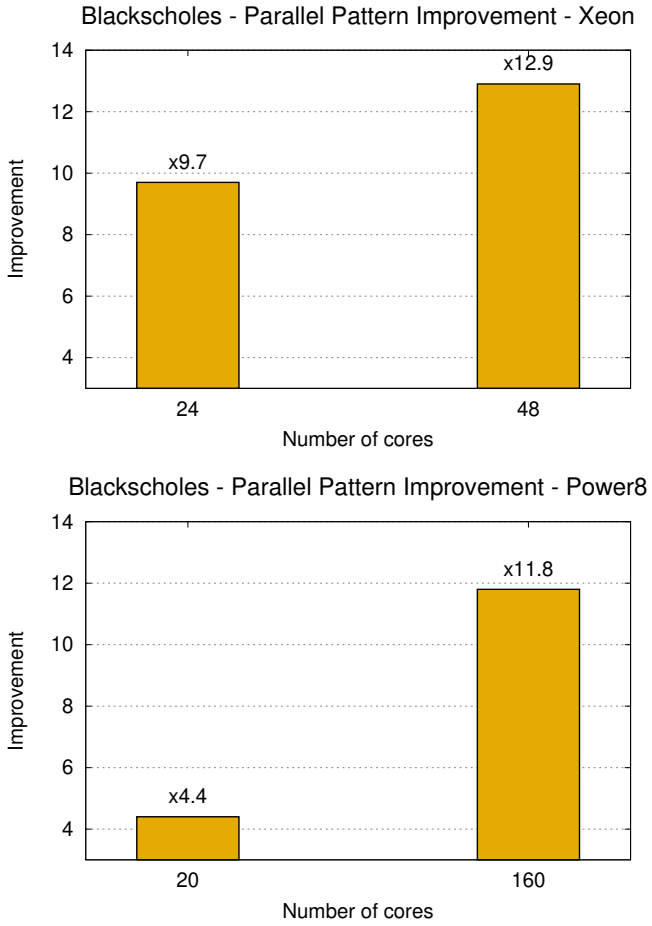


Figure 36: Improvement of the Parallel Patterns version compared to the "pure" Actor Model version of the `blackscholes` benchmark on the `Xeon` and `Power8` platforms.

Differently from the `quicksort` application, the `recMM` implementation in the SAVINA benchmarks does not use a "pure" Actor Model 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* Parallel Pattern which 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 4096×4096 elements and a cut-off value of 128×128). Then, we implemented the SAVINA version without any data sharing in a "pure" Actor Model fashion. The results obtained (not reported here for space reasons) show an im-

provement of the Parallel Patterns 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 (~18 on the `Xeon` and ~21 on the `Power8`), whereas the maximum speedup of the Parallel Patterns 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.

Also, we consider the `blackscholes` PARSEC benchmark, a real application that prices a portfolio of European options using the Black-Scholes partial differential equations [35]. This application can be parallelized iterating a fixed number of times a *Map* pattern [85]. We first implemented a "pure" Actor version of the *Map* parallel pattern which followed the *share-create* communication protocol (cf. Section 5.4). Then, we used the *Map* pattern presented in Section 7.2 for producing a second version. In the first version, the 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" Actor Model, it performs considerably worse than the one based on the *Map* Parallel Pattern, which internally makes more extensive use of the shared memory (Figure 36).

In the next subsection, together with other benchmarks, we compare the speedup of the Pattern-based `blackscholes` application against other parallel frameworks.

7.4.2 *Testing Actors+Parallel Pattern with some* PARSEC *benchmarks*

In this section, we study the Parallel Patterns-based parallelization of the PARSEC benchmarks selected. The performance results obtained are compared with those of the native PTHREADS and FASTFLOW implementations. We selected the FASTFLOW implementation because it uses the same pattern-based approach of the Parallel Patterns library, as described in De Sensi et al. [85]. Moreover, the FASTFLOW implementations have demonstrated comparable or better results on the PARSEC benchmarks with respect to other parallel library/framework implementations, e. g., INTELTBB, OPENMP. Table 5 summarizes the results obtained by using several different parallel libraries implementing the `blackscholes` applications of the PARSEC benchmark. We tested the benchmark on both the `Xeon` and the `Power8` platforms with parallelism degree equal to the number of physical cores and the total number of logical cores of the platform.

Blacksholes - Xeon

Parallel degree	FastFlow 2.1	OpenMP 2.0	Pthreads	SkePU 2	TBB	OmpSs 16.06.3	CAP+PP
24	6.61 s	8.49 s	8.44 s	8.19 s	7.31 s	6.72 s	6.11 s
48	4.69 s	4.64 s	4.59 s	4.83 s	5.07 s	4.86 s	4.58 s

Blacksholes - Power8

Parallel degree	FastFlow 2.1	OpenMP 2.0	Pthreads	SkePU 2	TBB	OmpSs 16.06.3	CAP+PP
20	8.94 s	11.76 s	10.25 s	11.65 s	9.83 s	8.68 s	10.33 s
160	4.97 s	5.62 s	5.97 s	4.62 s	4.67 s	5.93 s	4.91 s

Table 5: Results obtained considering several different parallel library/framework implementations for the `blackscholes` application [85].

FERRET

This application is based on the `ferret` toolkit [143] 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 *SeqActor* patterns). The first and last stages of the pipeline are instead *SeqActor* patterns reading and writing from/to the local disk. The same logical nesting of Parallel Patterns is used in the FASTFLOW version. Figure 37 shows the speedup of the `ferret` benchmark on the two platforms considered. The results obtained by using the Parallel Patterns 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 7.1. `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 main 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 parti-

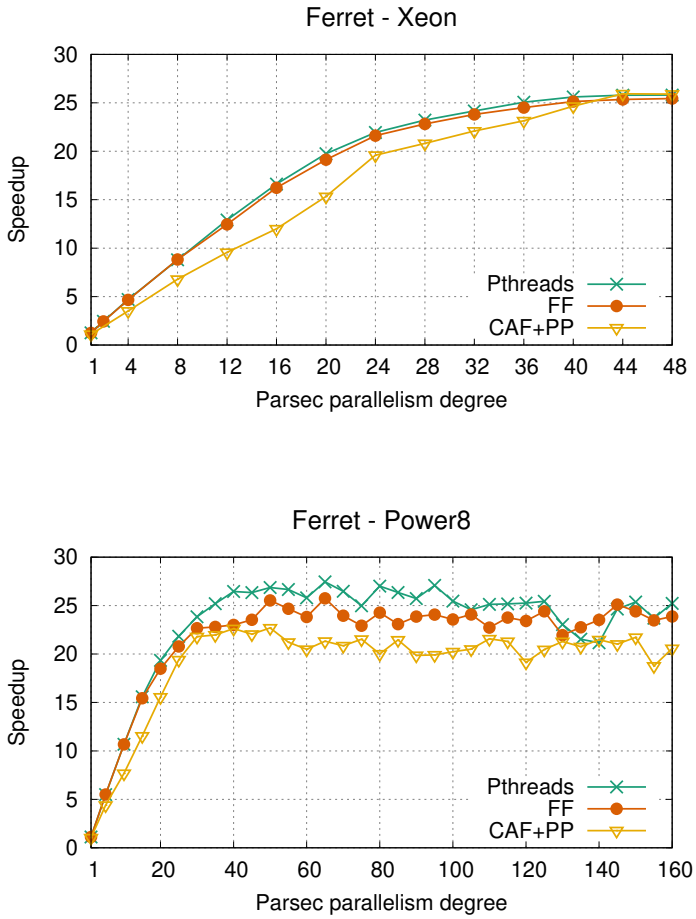


Figure 37: Speedup of the `ferret` benchmark of the PThreads, FastFlow (FF) and CAF plus Parallel Patterns (CAF+PP) implementations on the `Xeon` and `Power8` platforms considering as baseline the PThreads version with 1 thread.

tions while for `raytrace` a dynamic scheduling policy of the frame's partitions has to be used to obtain acceptable speedup.

Figure 38 shows the speedup of the `blackscholes` benchmark. The speedup of the Parallel Patterns 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 7.2).

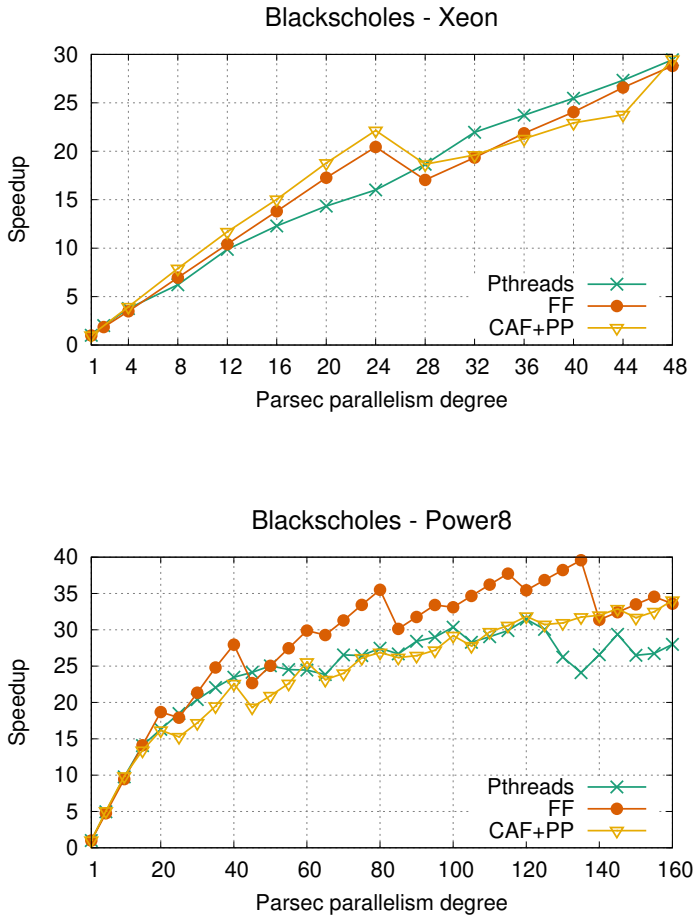


Figure 38: Speedup of the `blackscholes` benchmark of the PThreads, FastFlow (FF) and CAF plus PPs (CAF+PP) implementations on the `Xeon` and `Power8` platforms considering as baseline the PThreads version with 1 thread.

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 39 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 rout-

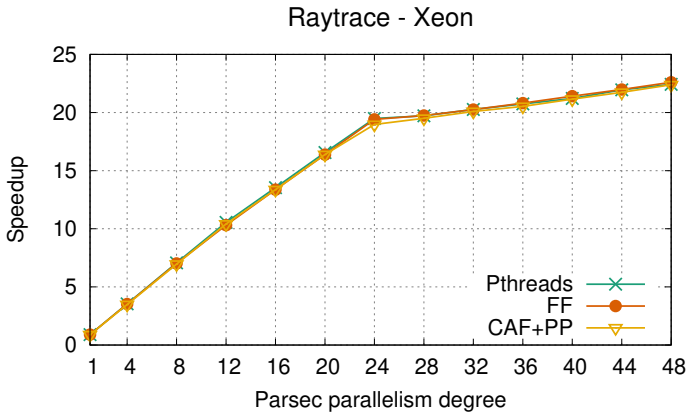


Figure 39: Speedup of the `raytrace` benchmark of the PTHREADS, FASTFLOW (FF) and CAF plus Parallel Patterns (CAF+PP) implementations on the `Xeon` platform considering as baseline the PTHREADS version with 1 thread.

ing cost, the algorithm 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 40 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.

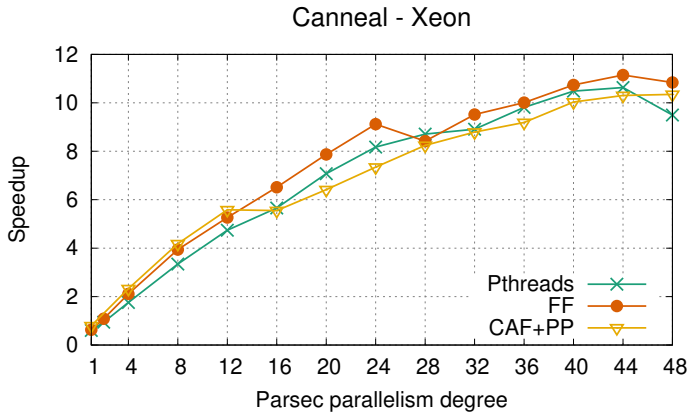


Figure 40: Speedup of the `canneal` benchmark of the PThreads, FastFlow (FF) and CAF plus Parallel Patterns (CAF+PP) implementations on the `Xeon` platform considering as baseline the PThreads version with 1 thread.

7.5 SUMMARY AND DISCUSSION

Employing the shared-memory programming paradigm on multi-/many-core architectures is of foremost importance to boost application performance. The Actor Model, with its strict memory isolation semantics, introduces extra overheads on this platform, particularly in data-parallel computations. We proposed to use Parallel Pattern abstractions on top of the Actor Model to introduce a software layer in which to inject platform-specific optimizations and well-known parallel structures.

In Section 7.1 we described the design guideline that we followed to achieve the integration between the Actor Model and the Parallel Patterns. We proposed an initial set of both data-parallel and control-parallel patterns. Although the data-parallel ones are those which most benefit from low-level shared-memory optimizations, the control-parallel provide the ability to compose patterns together to build more structure into the Actor-based applications. Section 7.3 and Section 7.2 describe the Parallel Pattern implementations on top of the C++ Actor Framework. Section 7.4 contains the evaluation of our approach using a set of well-known benchmarks coming from the Savina and Parsec benchmark suites.

To summarize the results, we observed that the Parallel Patterns proposed and implemented in the CAF framework seamlessly integrate with standard Actors and can be profitably used to speed up the performance-critical portions of the Actor-based application where the patterns can be introduced. The shared-memory abstraction confined within the skeleton implementation of the patterns, gives a significant performance boost to Actor-based applications compared

with "pure" Actor Model implementations. Our tests confirm that Actors+Parallel Patterns is a flexible and expressive parallel programming model capable of obtaining performance comparable to state-of-the-art specialized implementations on multi-/many-core platforms, without discarding essential features of the Actor Model such as memory isolation and the dynamic spawning of Actors.

8

HIGH-THROUGHPUT STREAM PROCESSING WITH ACTORS

Parts of this chapter was been published in the Workshop on Programming based on Actors, Agents, and Decentralized Control *(AGERE!) [177].*

In Chapter 7 we proposed the design and implementation of a set of Parallel Patterns, which synergically integrate with the Actor Model. Parallel Patterns can be composed and nested to build complex structures, in which *Pipeline* and *Farm* represent the infrastructure and *Sequential*, *Map* and *Divide&Conquer* are the core components encapsulating the business-logic code. The pattern compositions can be spawned and used by other Actors of the Actor-based application. We also demonstrated that it is possible to implement efficient data-parallel computations in Actor-based application on multi-/many-cores by leveraging our Parallel Patterns.

In this chapter, we focus on streaming computations and specifically on high-throughput streaming applications executed on multi-/many-cores. Actor-based languages and frameworks are more and more employed to design and develop complex streaming applications that need high flexibility, adaptivity, and high-scalability. However, in the Actor Model, the scalability concept is often associated with scale-out settings, i. e., large distributed systems or clusters. Nonetheless, solutions capable of consolidating several distributed servers in a single scale-up multi-/many-core server have recently gained special attention since they can reduce hardware costs, software licenses cost, data-center space, and power consumption [23]. To this end, the "pure" Actor Model does not offer enough room for enhancing its efficiency on a single scale-up multi-/many-core server. This is primarily due to the impossibility of introducing low-level optimizations in the messaging systems without breaking the model's semantics. Message-passing performance is crucial in streaming applications, where concurrent streaming entities (called *operators*) usually send a high number of small messages (called tuples in the Data Stream Processing domain). As we will demonstrate, the Actors' message-passing semantics may introduce significant performance degradations in high-throughput application contexts characterized by the exchange of many small messages.

In the following, we propose a set of optimizations of our Parallel Patterns to reduce these performance overheads. Therefore, in this chapter, we concentrate on *Control-parallel Patterns*, i. e., *Sequential*, *Pipeline*, and *Farm*.

The chapter starts describing the problems to adopt "pure" Actor Model implementation for the streaming computation (Section 8.1). We propose two microbenchmarks to assess the performance degra-

dation of C++ Actor Framework (CAF) and to be able to sharpen our optimization interventions. Then, Section 8.2 presents the sets of optimization implemented on top of our Parallel Patterns. Indeed, we propose a new skeleton implementation of the *Sequential*, which we called *SeqNode*. The *SeqNode* is capable of optimizing message-passing operations in pipeline structures made of compositions of Parallel Patterns. Moreover, we proposed a specific optimization for the *Farm* and *Pipeline* patterns, which reduces the number of message hops, and a simple backpressure mechanism to control memory consumption. Finally, we evaluate our approach by implementing a set of well-known streaming benchmarks (Section 8.3), comparing the results of the optimized Parallel Patterns with those implemented in Chapter 7.

8.1 MOTIVATIONS AND PROBLEM STATEMENT

The Actor Model is particularly suited to implement streaming applications because of the many concurrent entities usually involved, the explicit management of routing of messages, and the absence of shared states among operators. In the context of high-throughput demanding streaming applications targeting multi-/many-core servers, the use of system-level languages for implementing the Actor Model is unquestionably a performance plus compared to, for example, Java-based implementations. Indeed, several well-known implementations of the Actor Model, such as Erlang [24] and Akka [18], use virtual machine abstractions. Instead, C++ Actor Framework (CAF) applications are compiled directly into native machine code. However, CAF does not provide specific support for data-intensive streaming applications[1].

To evaluate the performance of CAF Actors in high-throughput streaming computations, we implemented a simple microbenchmark, i. e., a pipeline composed of three sequential stages. We built the benchmark using the Parallel Pattern abstractions we discussed in Chapter 7, and we compare the results obtained with those obtained by using a direct thread-based C++ implementation of the pipeline.

Specifically, we used the Parallel Pattern *Pipeline*, and connected three *SeqActor*s in a linear chain. The first Actor (called *Source*) generates a stream of tuples at the maximum speed possible (each tuple is 24 B). The second Actor (called *Forwarder*) forwards each input tuple to the next stage, and finally, the last Actor (called *Sink*) collects all tuples. Then, the same pipeline has been implemented by using three C++ threads and two FIFO lock-free queue [15] to implement the channels between the stages.

1 CAF offers experimental support for dataflow streams between Actors [194]. We did not use such a building-block in our implementation because, from the performance standpoint, it does not solve the issues outlined in this chapter

The two implementations were executed for 60 s and their throughput was measured in the Sink node. CAF (version `0.17.5`)[2] has been configured to run with three run-time Worker threads and with the *aggressive* polling strategy that maximizes system reactivity [211].

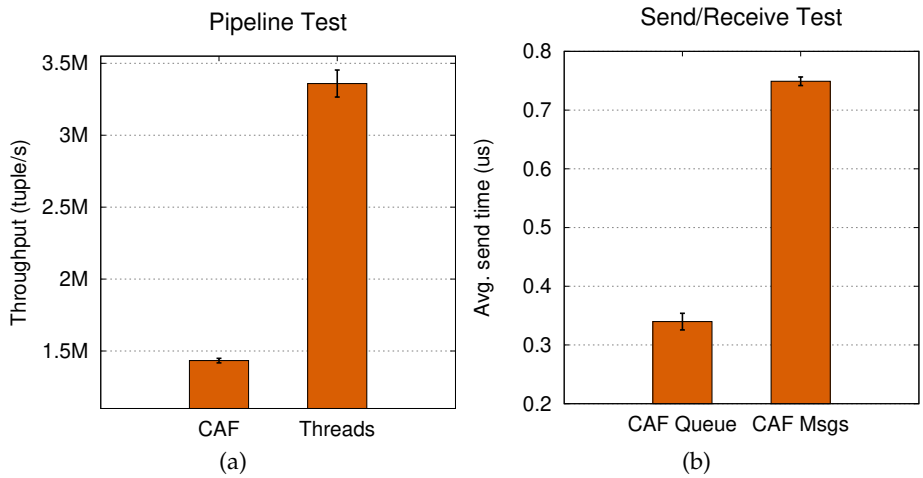


Figure 41: (a) Pipeline microbenchmark. CAF-based *vs* "manual" thread-based implementation. (b) Producer-Consumer microbenchmark. Simplified (CAF Queue) *vs* default CAF messaging system.

Figure 41a shows the results obtained. The measured throughput by employing CAF Actors is more than 2× lower than the thread-based implementation (1.4M *vs* 3.3M tuples/s).

To better understand the problem, we used a second microbenchmark implementing a simple Producer-Consumer pattern using two CAF Actors exchanging 100M small messages (4 B each). The objective is to evaluate the overhead for exchanging a single message between `Producer` and `Consumer`.

The first version uses the lock-free queue used in the CAF run-time for implementing Actors' mailboxes as a communication channel between `Producer` and `Consumer` (cf. Section 2.4.1). Therefore, the `Producer` pushes the messages directly into the queue while the `Consumer` pops them out.

In the second version, instead, we used the default CAF messaging system, which leverages the same lock-free queue with the dynamic message dispatching. Indeed, CAF adopts type-erasure to send dynamic messages to Actors. The original type of the message is annotated and then used in the pattern matching phase in the receiver Actor. After the Actor behavior is selected the message is cast back to the original type and the behavior lambda function is executed.

Figure 41b shows the results of this second microbenchmark. The overhead introduced by the Actors messaging system is more than 2× the base case (0.33 µs *vs* 0.75 µs). Such overhead is due to the com-

2 We also tested the pre-release `0.18.0` obtaining similar performance figures.

plexity of managing different types of messages, even if, as in our microbenchmarks, the Actors will always receive the same input type for the entire execution. This extra cost, perhaps, can be lowered by fine-tuning the implementation code, but certainly, it cannot be completely removed without losing the flexibility of defining multiple behaviors for an Actor. In streaming applications, operators work on statically defined input and output data types (the type of input and output tuples), and the extra complexity of managing multiple data types is the primary source of overhead.

For these reasons in the next section we propose a new skeleton implementation of the *SeqActor* which we called *SeqNode*. Using the *SeqNode* in combination with the *Pipeline* and *Farm* Parallel Pattern permits us to build a graph of concurrent entities. This graph statically defines their message types, thus removing the overhead and the complexity for dynamic dispatching.

8.2 STREAM-PARALLEL PATTERNS

In this Section, we discuss two optimizations of our Parallel Pattern implementations for the high-throughput streaming domain: a new *implementation skeleton* of the *Sequential* Parallel Pattern, and a specific optimization raising from the composition and nesting of *Farm* of *Pipeline* components. Also, we discuss a simple implementation of a backpressure mechanism to limit the number of messages produced by a sequential node in a *Pipeline* chain to avoid overloading a slower consumer node, i. e., a bottleneck node. Indeed, Actors use unbounded input mailboxes; therefore, in the context of high-throughput streaming of messages among Actors, it is probable that some slower Actors will have an explosion of messages in their input queue. Without a mechanism to limit the output data-rates of Actors, the Actor Model could not be used in these high-throughput contexts.

8.2.1 *Streaming Operators*

To tackle the message-passing overhead problem presented in Section 8.1, we propose a new *implementation skeleton* for the *Sequential* pattern capable of handling a single message type and optimized for defining high-throughput data streaming operators in CAF. Such new skeleton is implemented by the *SeqNode* class whose interface is inspired to the one provided by the *node* building-block in the FastFlow parallel library [16].

Listing 7 shows its current interface. Each new *operator* must define at least the `consume` method, which is called as soon as a tuple is available to be consumed by the node. During the *SeqNode* execution, input tuples are processed sequentially and in order. The other two methods `on_start` and `on_stop`, are automatically invoked

once when the node starts and right before it terminates, respectively. These virtual methods may be overwritten in the user-defined operator to implement initialization and finalization code for the operator node. The *SeqNode* has a typed input queue and zero, one or more typed output queue references (implemented by the `multiQueues` object in Line 13 of Listing 7) in order to connect the operator implemented by the node to one or more different *SeqNode*s.

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

Listing 7: The base interface of the *SeqNode* streaming pattern.

```
1  struct Operator : SeqNode<T> {
2    Operator(MyState s) : SeqNode{}, s_{s} {}
3    void consume(T &x) override {
4      T y = do_something(x, s_);
5      send_next(std::move(y));
6    }
7
8  private:
9    MyState s_; // operator local state
10 };
11
12 int main() {
13   // ...
14   MyState s1, s2;
15   Operator op1{s1};
16   Operator op2{s2};
17   Pipeline pipe{op1, op2};
18
19   auto p = spawn_pattern(sys, pipe).value();
20   // ...
21 }
```

Listing 8: Simple example showing how to define a *SeqNode* and how to use it within *Pipeline* pattern.

The *Sequential* pattern has two implementation skeletons: *SeqActor* and *SeqNode*. The main differences between them are that the first behaves like a standard CAF Actor, and so it can exchange messages

with any other Actors being them a standard CAF Actor or a Parallel Pattern. On the contrary, the *SeqNode* can be used only inside a *Pipeline* pattern, or it can be a Worker of a *Farm* pattern. The *Pipeline* and *Farm* patterns provide the necessary interface to enable the *SeqNode* to communicate with other standard Actors. Listing 8 shows how to define *SeqNode* operators and how to connect them in a *Pipeline* pattern.

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

8.2.2 *Pipeline compositions of Farms*

The functional-style composition is one of the primary features of the proposed Parallel Patterns. Streaming applications can be easily modeled by one or more *Pipeline* compositions of *SeqNode*s where some operators are replicated using the *Farm* pattern and suitable distribution policies. The *Farm* implementation skeleton uses an *Emitter*, which is in charge to distribute the incoming messages to the Worker executing the selected distribution policy (cf. Section 7.3). The Emitter introduces an extra message hop for each operator replica in a streaming network formed by multiple *Farm* compositions. Clearly, these extra hops may impact the end-to-end latency for traversing the entire network of operators and may introduce a bottleneck in case of high data rates and fine-grained operators. For these reasons, we decided to introduce a new implementation skeleton for the composition of *Farm* PPs within a *Pipeline* to reduce the number of Emitters hence the number of message hops.

When a *Pipeline* pattern connects two consecutive *Farm*s (regardless of the kind of PP compositions used in the Workers), the Emitter of the second farm is automatically removed, and its distribution policy is implemented within the *send_next* method of the right-most leaf PPs (i. e., *Sequential*, *Map*, and *Divide&Conquer*) present in all Worker replicas of the first *Farm*. Figure 42 shows a simple use-case in which three *Farm* patterns are used within a *Pipeline* to replicate three *Sequential* operators (in the figure, we used *SeqNode*s). Only the first *Farm* preserves the Emitter because it can receive messages from outside the pattern, e. g., from a standard CAF Actor. On the contrary, the second and third *Farm*s do not have the Emitter.

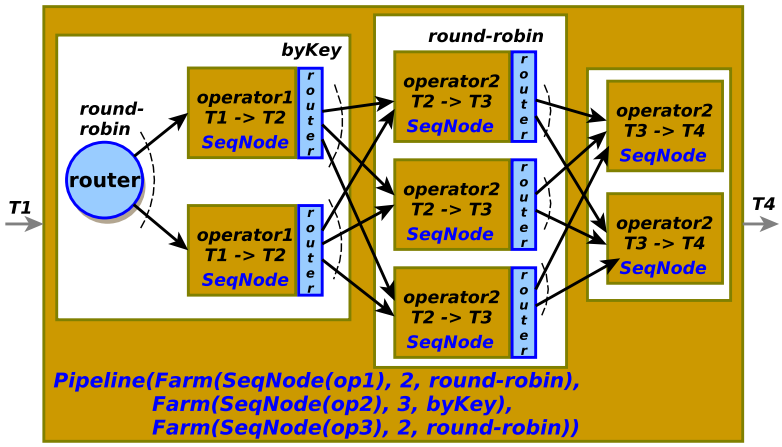


Figure 42: Optimized *Pipeline* composition of three *Farm*s patterns running replicas of *SeqNode* operators.

8.2.3 *The* `send_next_if` *primitive.*

In almost all Stream Processing Engines (SPE), e. g., Apache Storm [22] and Apache Flink [20], a form of backpressure is implemented to guarantee that sender Actors cannot overload receiver Actors due to different relative speed [136]. Instead of implementing complex and costly demand signaling protocols between each sender and receiver, we decided to implement a more straightforward form of flow-control by providing a `send_next_if` command within Parallel Patterns. The *if* condition is applied to the actual number of messages present in the destination queue. If the length of the queue (observed by the sender) is greater than the specified parameter value, the send command immediately returns to allow the user to take the suitable actions (e. g., waiting a while before retrying or discarding the message or buffering it locally).

The `send_next_if` command, like `send_next`, uses the policy configured by the next pipeline stage to select the queue in which to insert the message. However, if the next stage policy is either *round-robin* or *random*, the command will try to enqueue the message in all next queues before returning with failure. This simple mechanism allows the sender to implement flow-control strategies and to auto-regulate the speed at which messages are sent. In high data rate scenarios, such control-flow strategies are typically needed only in the *Source* operator.

We modified the CAF LIFO queue (cf. Section 2.4.1) by adding a new method that returns the estimated length of the queue (i. e., `synchronized_size`). This method will be internally used by the `send_next_if` command to check the queue length of *SeqActor*s and *SeqNode*s. `synchronized_size` counts the elements present both in the multiple FIFO queues and in the LIFO thread-safe queue. In the first case, the calculation is performed without synchronization and

might return an approximated value. Instead, the LIFO queue count is performed within a spin-lock to avoid data-races. However, in the implementation we made, the extra cost of the synchronization is paid only in the `send_next_if` command where the estimation of the queue length is needed and never paid in the `send_next` command.

8.3 EVALUATION

In this Section, we consider again the *Pipeline* microbenchmark already presented in the Section 8.1, and we also consider a set of streaming applications typically used to evaluate Stream Processing Engines.

All experiments were conducted on the Xeon server (the complete specification is in Section 5.1). The CAF version used is the `0.17.5`, and for all tests, the number of run-time Worker threads was set to the total number of Actors used. The Work-Stealing CAF run-time has been configured to use the most *aggressive* polling strategy to maximize reactivity of threads.

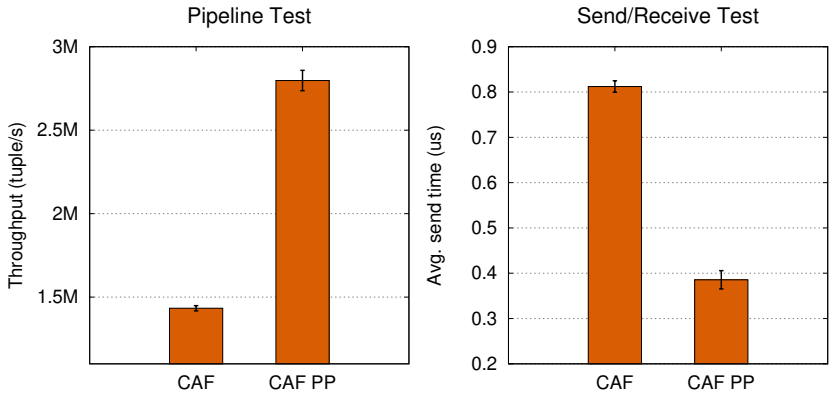


Figure 43: (a) Maximum throughput of the *Pipeline* microbenchmark and (b) the average send time of Producer-Consumer microbenchmark.

Our first test was to evaluate the improvement of using *SeqNodes* instead of *SeqActors* in the three-stage pipeline microbenchmark. Figure 43a compares the maximum throughput (tuples/s) obtained by the the two versions, CAF implementing *Pipeline* (*SeqActor*,*SeqActor*,*SeqActor*); CAF PP implementing *Pipeline* (*SeqNode*,*SeqNode*,*SeqNode*), of the same operators. A large part of the overhead discussed in Section 8.2 has been removed. The difference with the baseline (called *Thread* in Figure 41a), is now about 15% (2.8M *vs* 3.3M tuples/s), which means that there is still a small margin for fine-tuning our *SeqNode* implementation. Concerning the cost for exchanging a message between a Producer and a Consumer

Actor, Figure 43b shows that the *SeqNode*-based implementation (i. e., CAF PP) reduces the average time from 0.81 µs to 0.38 µs.

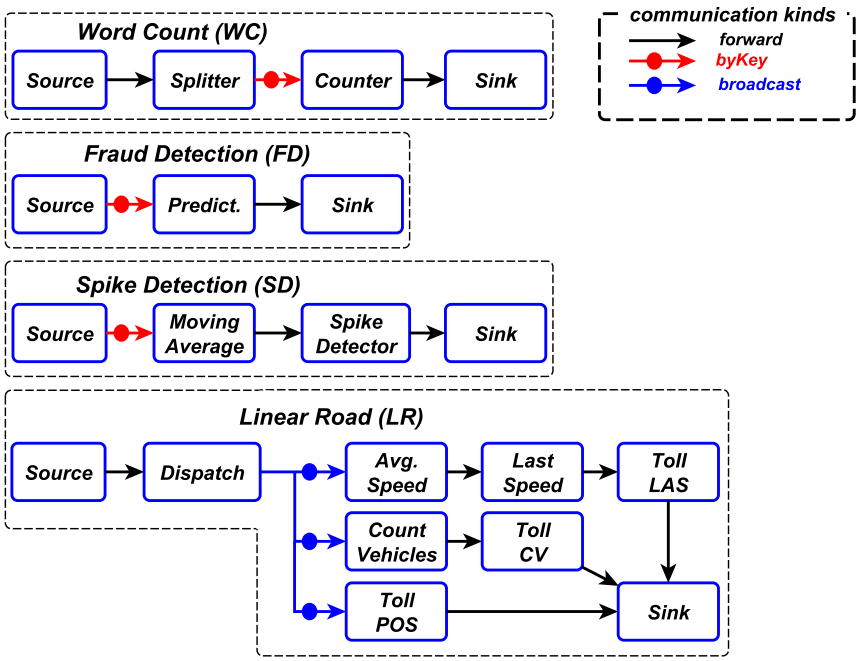


Figure 44: Applications used in the evaluation.

We now consider four well-known streaming applications[3] whose operators graphs are sketched in Figure 44, namely:

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

The figure also shows the kind of communication among operators: *forward*, *byKey* and *broadcast*. These communication attributes are meaningful if the next operator is replicated using a *Farm* pattern.

3 The C++ source code is publicly available in GitHub: `https://github.com/ParaGroup/StreamBenchmarks`.

The *forward* communication is the default one. It states that an input tuple can be assigned to any replicas. We implemented this communication mode with the *round-robin* policy strategy of the *Farm* pattern.

The *byKey* distribution allows sending all input tuples with the same key attribute (i. e., a specific field of the tuple) to the same operator replica.

Finally, the *broadcast* distribution duplicates the tuple and sends it to all next operators.

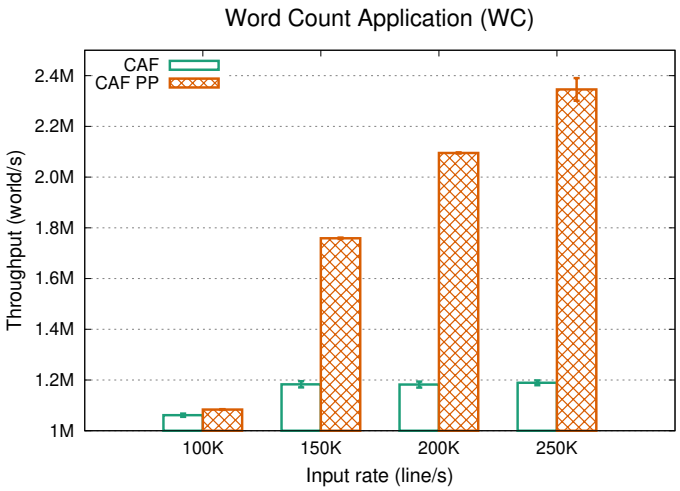


Figure 45: Throughput expressed in words per second, varying the input rate for the *Word Count* application.

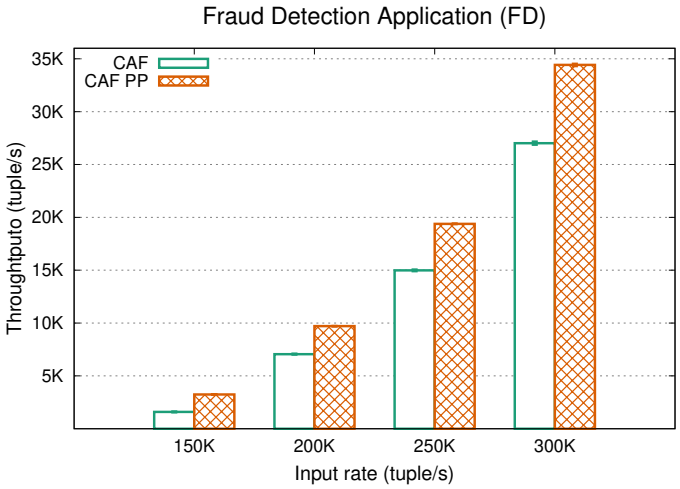


Figure 46: Throughput expressed in tuples per second varying the input rate for the *Fraud Detection* application

In Figure 45, 46, 47, 48 we showed the throughput obtained by implementing the applications' operators (one replica per operator) by using the two implementation skeletons for the *Sequential* pattern (*SeqActor vs SeqNode*, labeled with CAF and CAF PP, respectively). All tests have been executed 20 times for 60 seconds. The average value

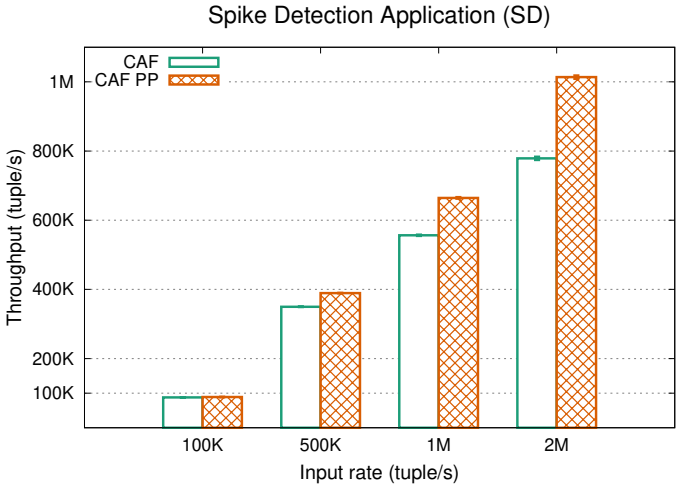


Figure 47: Throughput expressed in tuples per second varying the input rate for the *Spike Detection* application

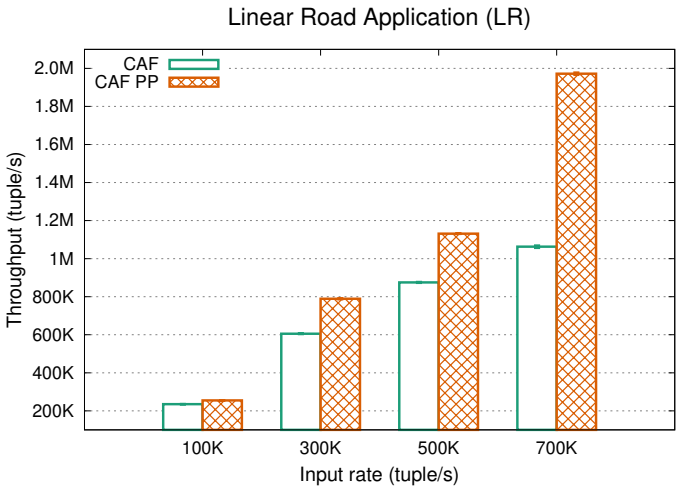


Figure 48: Throughput expressed in tuples per second varying the input rate for the *Linear Road* application

obtained and the error-bar are shown in the figures. The throughput has been measured in the *Sink* operator varying the input data rate in the *Source*. As long as the data rate is relatively low, there are no significant differences between the two implementation skeletons, even though the end-to-end latency is higher for the *SeqActor*-based implementation (cf. Table 6).

Conversely, with high data rates, the difference between the two versions is significant.

Word Count presents the biggest difference. This is due to the very high data rate produced by the `Splitter` operator that gets in input a line from the `Source`, and produces in output all words it contains, thus multiplying the nominal input data rate.

Figure 49 also shows the performance improvement obtained for all applications by the *SeqNode*-based implementations (CAF PP vs

	WC	FD	SD	LR
CAF	536 µs	454 µs	461 µs	531 µs
CAF PP	24 µs	5 µs	2 µs	24 µs
Improvement	22	84	283	22

Table 6: Applications latency with rate 10K tuples/s. The improvement is the ratio between CAF PP and CAF.

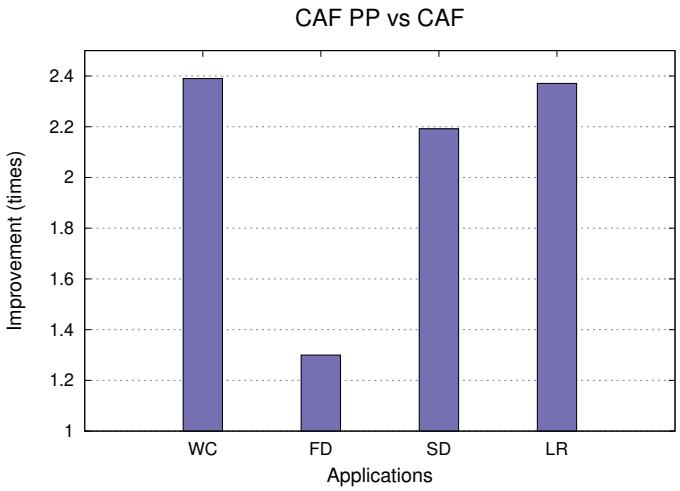


Figure 49: Performance improvement of CAF PP vs CAF for WC, FD, SP, LR applications. The tests were executed allowing the *Source* operator to generate at the maximum rate.

CAF). In this case, for both implementations, we ran the *Source* operator at maximum speed. In addition, they use the `send_next_if` primitive. The "queue length value" is set to 1K elements, and if reached, the *Source* waits for some nanoseconds before retrying the send, thus implementing a very aggressive generation of data without increasing the size of receivers' input queues indefinitely. In this way, the internal pipeline operators are not overloaded, and the entire system stabilizes after a few seconds of execution. In this case, we executed all applications for 10 minutes.

As expected, the improvement of the CAF PP version using *SeqNodes* is significant, more than 2× in all application but *Fraud Detection* where the improvement is about 30%.

Finally, Table 7 shown the results obtained in all the applications by replicating all the streaming operators by using the *Farm* pattern. Instead of finding the best replication degree for each operator (operators have different execution times, and some of them may need more replicas while others do not need to be replicated at all), we decided to equally replicate all of them until we fill up all the logical cores of the server considered (48 in our case).

The replication degree is specified in the table. In this way, we can evaluate the behavior of the two implementation skeletons under very

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

Table 7: Applications throughput (tuples/s) obtained replicating all operators. The improvement is the ratio between CAF PP and CAF.

high data rates, since all *Source* replicas execute at maximum speed. In this test we used for both versions the implementation skeleton for the *Farm* pattern that removes the Emitter between two consecutive *Farm* PPs as described in Section 8.2.2. For all applications, we obtained a performance improvement in terms of throughput, and as expected, the relative distance between the two implementation skeletons (i. e., *SeqActor vs SeqNode*) further increased in all applications but *Linear Road*, where the relative distance remains about the same due to its peculiarities.

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

8.4 SUMMARY AND DISCUSSION

Actor-based languages and frameworks are more and more employed to design and develop complex streaming applications that need high flexibility, adaptivity, and high-scalability. However, when targeting a single scale-up multi-/many-core system capable of consolidating multiple distributed servers, the "pure" Actor Model approach introduces extra overhead, impairing its usage.

To assess the Actor Model performance figures in the context of high-throughput streaming applications, we conducted a careful performance analysis employing both simple microbenchmarks as well as a set of well-known streaming applications typically used to assess the performance of Stream Processing Engines (SPE). We identified two main issues. First, the complexity of the messaging system for managing different message types in an Actor is a limiting performance factor in Actor-based streaming applications where streaming operators usually deal with a single data type. Second, at high input data rates, the Actors mailboxes having unlimited capacity pushes too much pressure in the memory system, making it challenging to stabilize the application behavior and limit memory consumption.

To tackle these issues, we proposed a set of optimizations for our Parallel Patterns targeting streaming computations on multi-/many-

cores. In particular, we introduced the *SeqNode* implementation skeleton to be used within *Farm* and *Pipeline* Parallel Patterns. Differently from the *SeqActor*, discussed in Section 7.3.1, the *SeqNode* is a specialized Actor capable of handling only statically-defined single input and single output message types. Using the *SeqNode* in combination with the *Pipeline* and *Farm* patterns enables to build streaming graphs of statically defines types, thus removing the overhead and the complexity associated with the dynamic dispatching of messages used in plain Actors. Other optimizations introduced are related to a new implementation skeleton of the *Farm* pattern that optimizes *Pipeline* compositions of consecutive *Farm* patterns to reduce the number of message hops. Moreover, we discussed the issues of the Actors' unbounded mailboxes in streaming computations, and we introduce a simple backpressure mechanism for *SeqNode* operators leveraging a new communication primitive that takes into account the queue length of the receiving Actors.

The evaluation conducted demonstrates that the optimizations introduced significantly enhance message-passing communications, improving Actor-based streaming applications' overall performance. Moreover, it also demonstrates that the whole set of patterns proposed is expressive enough to implement a significant set of well-known streaming benchmarks.

Part III

SUMMARY

9

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

9.1 SUMMARY

With the large diffusion of multi-/may-core systems, several different programming models have been proposed targeting such platforms. All of them, in different ways, aim at overcoming the limitations of the traditional shared-memory programming approach based on threads and locks. Unfortunately, most of the currently available parallel programming frameworks are either not high-level enough to be profitably used by the standard programmers or not flexible enough to be used in different applications domains with strong parallelism requirements (e. g., data-streaming, data-intensive parallelism, task-parallelism).

The Actor Model is an elegant concurrency model with important memory safety guarantees, which prevents common bugs and security vulnerabilities when dealing with concurrency. It offers high flexibility for building communication topologies composed of concurrent Actors. This programming model is gaining increasing success in non-performance critical scenarios thanks to its simplicity and flexibility. Actor-based applications are characterized by *unstructured* and not rigid communication graphs where Actors can be created dynamically. Each Actor can spawn other Actors and can communicate with them just by using their references. Unluckily, the Actor Model does not provide the non-expert parallel programmer with high-level components for implementing recurrent parallel problems. Moreover, its strict message-passing semantics does not permit the introduction of essential optimizations for shared-memory platforms, making the creation of parallel libraries a non-trivial task, especially for data-parallel computations.

The thesis describes our research work to combine the Actor Model with the structured parallel programming approach based on Parallel Patterns, i. e., well-defined ready-to-use parallel components suitable to be used in recurrent parallel problems (e. g., *Pipeline*, *Map-Reduce*, *Divide&Conquer*). The implementation skeletons of those components can be optimized for a given platform, and thus they can also be used to inject platform-specific optimizations otherwise prevented by the Actor Model semantics.

Parallel Patterns alone might limit the freedom of the programmers for expressing the parallelism in their applications. It may also happen that available Parallel Patterns do not support the required paral-

lel forms needed to solve the programmer's problem. In this respect, we believe that a synergic integration of Actors and Parallel Patterns represents the right balance between flexibility, programmability, expressiveness, and performance on multi-/many-core systems.

In this thesis, we discussed some limitations of the "pure" Actor Model, both related to programmability and performance aspects on multi-/many-cores, and we described our efforts to demonstrate the viability of combining Actors and Parallel Patterns and the leverage effect produced by their integration.

In Chapter 5 we focused on studying the memory isolation property. The Actor Model enforces the memory isolation property of the Actors' internal state to avoid potential subtle data-races. This property is somehow in contrast with some shared-memory-based optimizations that are crucial to efficiently implement data-parallel computations on physical shared-memory systems.

In Chapter 6 we described our first attempt to combine the Actor Model with Parallel Patterns through a software accelerator for data-parallel computations. The accelerator could be installed on a separated portion of the CPU core resources by using our implementation in the C++ Actor Framework (CAF) of the *thread-to-core-affinity* feature. All Actors can then use the accelerator to speed up data-parallel computations on large incoming messages.

We then developed the full integration of CAF Actors with a set of Parallel Patterns in a new parallel library called CAF-PP[1]. Each Parallel Pattern of the library is a "macro Actor". It can be spawned like standard Actors and can interact with them by using the same messaging system.

Chapter 7 and Chapter 8 contain the details of the implementation of the Parallel Patterns in the CAF-PP library. Specifically, in Chapter 7 we concentrated mainly on *Data-parallel Pattern*, whereas in Chapter 8 we focused on optimizing *Control-parallel Pattern* and their composition to efficiently implement high-throughput streaming computations.

We evaluated the performance of our Actors+Patterns library by using an extensive set of benchmarks on different multi-/many-cores platforms. Specifically, through a set of well-known parallel applications coming from the Parsec benchmark suite, we demonstrated that the Actor Model alone cannot fully exploit the computation capabilities of modern multi-/many-core platforms, especially for data-parallel computations. Instead, by leveraging our CAF-PP library atop the C++ Actor Framework (CAF) framework, we demonstrated that it is possible to achieve the comparable performance of raw PThreads-based, and thread-based specialized libraries (e. g., FastFlow) implementations without breaking the Actor Model design principles and semantics. We also considered a

1 https://github.com/ParaGroup/caf-pp

different class of applications for which the Actor Model is increasingly used: data streaming computations. In particular, we focused on high-throughput computations with high-performance requirements. Again, we demonstrated that Actors alone could not achieve high-throughput and low-latency in highly-demanding streaming computations. Instead, leveraging the optimized *Control-parallel Patterns* of our CAF-PP library, we demonstrated a performance improvement of more than 2× in terms of throughput and a significant reduction of the end-to-end message latency of a significant set of well-known streaming benchmarks.

9.2 CONCLUDING REMARKS

The limited expressivity and flexibility of current high-level parallel programming models targeting multi-/many-cores led us to propose and implement a new parallel library on top of the C++ ACTOR FRAMEWORK called CAF-PP. It results from a careful arrangement of two distinct parallel programming approaches, namely, Actor-based and Pattern-based parallel programming.

Several existing models suffer from being specialized for a given class of parallel problems, forcing the programmers to rethink or adapt their applications to use the selected model. In this respect, a paradigmatic example is the well-known *Google MapReduce* programming approach [87], and its open-source implementation HADOOP [21]. The framework was pushed by several programmers and researchers to be used to implement many different kinds of applications. Specifically, several graph-based algorithms were ported to this model. However, those implementations have been recently outperformed by *Vertex-Centric* programming models (e. g., GRAPHLAB) both concerning programmability and absolute performance figures [150].

Defining a parallel programming approach that is simple, malleable, efficient, and fully expressive is of paramount importance for building durable applications for current and forthcoming multi-/many-core systems. Parallel Patterns are practical software components useful to solve a given problem efficiently. They can be combined to solve a larger class of applications, even though it is not always possible to have all Parallel Patterns needed for all kinds of applications. In this scenario, the Actor Model represents a safe escape strategy. It offers a well-defined and robust model, providing the programmer with enhanced flexibility and memory-safety on multi-/many-cores.

The combination of these two programming approaches allows more choices for the programmers. In principle, by adopting our CAF-PP library, the programmers may even refrain from using Parallel Patterns for designing a given part of their application. If this is

the case, they can count on standard Actors, which are well-defined concurrent entities with clear semantics and strong memory-safety guarantees. Conversely, suppose the programmers discover that one or more Actors are potential bottlenecks for their applications. In that case, they can count on well-defined parallel components that can be used as "macro Actors" to parallelize the bottleneck Actors according to the parallel semantics they want to use (e. g., *Map-Reduce* or *Pipeline*). Even in this case, with the model exposed by the CAF-PP library, this operation can be done straightforwardly without changing or forcing the whole model's semantics.

Such a parallel programming view can be considered a general approach that combines highly specialized components to solve a given class of problems and a general-purpose and safe programming model to accommodate more specific users' needs. We believe that the resulting combination of Actors and Parallel Patterns may offer a new coherent approach to parallel programming, even for non-expert parallel programmers, maintaining a good trade-off between vertical specializations and general-purpose flexibility.

9.3 FUTURE PERSPECTIVES

This thesis describes our vision of a flexible, efficient, and memory-safe programming model for multi-/many-core systems. Most of the research work points to practical implementations and optimizations we defined for the smooth integration between Actors and Parallel Patterns.

We foresee incremental improvements to the library stability and the full integration with the C++ ACTOR FRAMEWORK framework. We also planned to include new Parallel Patterns to the current set, specifically those targeting data-parallel computations such as iterative *stencil*-based computations [12] used in many image and numerical data processing algorithms.

We conducted the experimental evaluation focusing on the performance enhancement that the Parallel Patterns we designed provides to the Actor Model. Therefore, we concentrated on the critical parts of the Actor-based applications for which we can provide substantial performance improvement through our patterns. We plan to study larger and more complex Actor-based applications to evaluate more in-depth the programmability advantages that our model can bring.

The work perspective is to back the more practical approach of this research with the formal definition of the proposed programming model. Actor Model has a long history of theoretical works that studied different aspects of Actor behavior building complete formalizations [8, 121]. We envision a formalization that includes the cooperation of the Actor Model with the structured parallel programming model based on Parallel Patterns. For instance, we could follow the

approach of Hains et al. [115, 139] in which the authors combines the BSP and Active Objects proposing a formal operational semantics.

Parallel Patterns offer the ability to reason about the parallel application structure at a high-level of abstraction and analyze the expected performance. Recently, RPLSH [108] propose a DSL-based toolchain supporting the design of parallel applications where parallelism is expressed via proper composition and nesting of Parallel Patterns. RPLSH guides the programmer to select the most suitable parallel abstraction based on the estimated performance by using per-pattern parametric cost models. It also offers automatic Parallel Patterns transformations and optimizations. At the end of the design process, the RPLSH generates the concurrent C++ code leveraging the FASTFLOW library. An interesting future extension of the work presented in this thesis is to include our set of Parallel Patterns within the RPLSH providing the users with the possibility to directly measure the advantages of using Parallel Patterns composition instead of plain Actors in Actor-based applications.

This research also opens some questions about other parallel programming models beyond the Actor Model. The Task-based parallel programming approach [208] is taking more and more attention, especially in the industry. The most evident advantage of using Tasks is the straightforward approach provided to the application programmer. The programmer has to be aware of how Tasks are defined with their input dependencies and how to create a Task. Then, the underlining task-engine is in charge to automatically distribute ready tasks to multiple concurrent worker threads. This high flexibility could be used to build either data-parallel or streaming applications[2]. Although Actor Model usually uses an engine to execute ready Actors, Tasks have a shorter lifespan than Actors, which could improve load balancing in some cases. However, Actors could be considered repeating tasks, i. e., every time an Actor receives a new message, a new task is ready to be computed. In a more comprehensive picture, we can add to our evaluation also the Thread-based model, in which threads have an even longer lifespan than Actors. Indeed, it could be interesting to compare Tasks, Actors, and Threads as three ways to decompose an application in a set of concurrent activities with different granularities and lifecycles, and studying the pros and cons of the three approaches considering both performance and programmability issues. We envision that Actors might provide the right trade-off to the programmer, being them capable of behaving either as Tasks or as Threads.

Finally, the Active Objects [78] has a similar approach to concurrency than Actors, and it provides clearer Object-Oriented functionalities than Actors [62, 181]. We used Actors instead of Active-Objects

2 However it was recently observed that Task-based *Stream Process Engines* (SPE) could be susceptible of higher tuple latency than thread-based SPE [151]

because of their communication mechanism based on the *send* primitive, which we believe provides more opportunities for easy integration with Parallel Patterns. Notwithstanding, it is also worth investigating how the Active Objects and Parallel Patterns programming approach can be combined in a unified model and understand whether the resulting model provides further expressivity and programmability cues than Actors+Parallel Patterns.

BIBLIOGRAPHY

[1] *2nd Gen AMD EPYC™ Processors | EPYC™ 7002 Series | AMD*. URL: https://www.amd.com/en/processors/epyc-7002-series (visited on 10/04/2020).

[2] *ABCL: Actor-Based Concurrent Language*. URL: https://en.wikipedia.org/wiki/Actor-Based_Concurrent_Language (visited on 06/03/2020).

[3] *Acteur-rs library*. URL: https://github.com/DavidBM/acteur-rs (visited on 06/03/2020).

[4] *Actix: Actor framework for Rust*. URL: https://github.com/actix/actix (visited on 06/03/2020).

[5] *Actor4j actor-oriented Java framework*. URL: https://github.com/relvaner/actor4j-core (visited on 06/03/2020).

[6] *Actr: Simple, fast and typesafe Java actor model*. URL: https://github.com/zakgof/actr (visited on 06/03/2020).

[7] Gul Agha. "Actors: A Model of Concurrent Computation in Distributed Systems." PhD thesis. 1984.

[8] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. "Towards a theory of actor computation: Extended abstract." In: *CONCUR '92*. Vol. 630. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 565–579. ISBN: 978-3-540-55822-4. DOI: 10.1007/BFb0084816.

[9] *Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala | Akka*. URL: https://akka.io/ (visited on 06/03/2020).

[10] Baseem A. AL-Twajre. "Performance Analysis of Messages Queue in the Different Actor System Implementation." In: *2019 XIth International Scientific and Practical Conference on Electronics and Information Technologies (ELIT)*. 2019 XIth International Scientific and Practical Conference on Electronics and Information Technologies (ELIT). Lviv, Ukraine: IEEE, Sept. 2019, pp. 127–131. ISBN: 978-1-72813-561-8. DOI: 10.1109/ELIT.2019.8892329.

[11] Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. "May-Happen-in-Parallel Analysis for Actor-Based Concurrency." In: *ACM Transactions on Computational Logic* 17.2 (Mar. 28, 2016), pp. 1–39. ISSN: 1529-3785, 1557-945X. DOI: 10.1145/2824255.

[12] M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, and M. Torquati. "A parallel pattern for iterative stencil + reduce." In: *The Journal of Supercomputing* 74.11 (Nov. 1, 2018), pp. 5690–5705. ISSN: 1573-0484. DOI: 10.1007/s11227-016-1871-z.

[13] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Design patterns percolating to parallel programming framework implementation." In: *International Journal of Parallel Programming* 42.6 (Dec. 1, 2014), pp. 1012–1031. ISSN: 1573-7640. DOI: 10.1007/s10766-013-0273-6.

[14] Marco Aldinucci and Marco Danelutto. "Skeleton-based parallel programming: Functional and parallel semantics in a single shot." In: *Computer Languages, Systems & Structures* 33.3 (2007), pp. 179–192. ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2006.07.004.

[15] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. "An Efficient Unbounded Lock-Free Queue for Multi-core Systems." In: *Euro-Par 2012 Parallel Processing*. Ed. by Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 662–673. ISBN: 978-3-642-32820-6.

[16] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Fastflow: High-Level and Efficient Streaming on Multicore." In: *Programming multi-core and many-core computing systems*. John Wiley & Sons, Ltd, 2017, pp. 261–280. ISBN: 978-1-119-33201-5.

[17] Christopher Allen and Julie Moronuki. *Haskell Programming from first principles*. Gumroad, 2017.

[18] Jamie Allen. *Effective Akka: Patterns and Best Practices*. " O'Reilly Media, Inc.", 2013. ISBN: 1-4493-6007-6.

[19] *AmbientTalk*. URL: http://soft.vub.ac.be/amop/ (visited on 06/03/2020).

[20] *Apache Flink*. 2020. URL: https://flink.apache.org/ (visited on 08/19/2020).

[21] *Apache Hadoop*. URL: http://hadoop.apache.org/ (visited on 12/29/2020).

[22] *Apache Storm*. 2020. URL: http://storm.apache.org/ (visited on 08/19/2020).

[23] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. "Scale-up vs Scale-out for Hadoop: Time to Rethink?" In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. event-place: Santa Clara, California. New York, NY, USA: Association for Computing Machinery, 2013. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523629.

[24] Joe Armstrong and "Ericsson Telecom Ab". "The development of Erlang." In: *Association for Computing Machinery* 32.8 (1997), pp. 196–203. DOI: 10.1145/258948.258967.

[25] Dominik Aumayr, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. "Asynchronous snapshots of actor systems for latency-sensitive applications." In: *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes - MPLR 2019*. the 16th ACM SIGPLAN International Conference. Athens, Greece: ACM Press, 2019, pp. 157–171. ISBN: 978-1-4503-6977-0. DOI: 10.1145/3357390.3361019.

[26] Keyvan Azadbakht, Frank S. de Boer, and Vlad Serbanescu. "Multi-Threaded Actors." In: *Electronic Proceedings in Theoretical Computer Science* 223 (Aug. 10, 2016), pp. 51–66. ISSN: 2075-2180. DOI: 10.4204/EPTCS.223.4.

[27] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. "SkIE: A heterogeneous environment for HPC applications." In: *Parallel Computing* 25.13 (1999), pp. 1827–1852. ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(99)00072-1.

[28] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. "P3L: A structured high-level parallel language, and its structured support." In: *Concurrency: Practice and Experience* 7.3 (1995), pp. 225–255. DOI: 10.1002/cpe.4330070305.

[29] Saman Barghi and Martin Karsten. "Work-Stealing, Locality-Aware Actor Scheduling." In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Vancouver, BC: IEEE, May 2018, pp. 484–494. ISBN: 978-1-5386-4368-6. DOI: 10.1109/IPDPS.2018.00058.

[30] *Bastion library*. URL: https://bastion.rs/ (visited on 06/03/2020).

[31] David Alessandro Bauer and Juho Makio. "Hybrid Cloud – Architecture for Administration Shells with RAMI4.0 Using Actor4j." In: *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*. 2019 IEEE 17th International Conference on Industrial Informatics (INDIN). Helsinki, Finland: IEEE,

July 2019, pp. 79–86. ISBN: 978-1-72812-927-3. DOI: 10.1109/INDIN41052.2019.8972075.

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

[33] Shuvra S. Bhattacharyya, Johan Eker, Jörn W. Janneck, Christophe Lucarz, Marco Mattavelli, and Mickaël Raulet. "Overview of the MPEG Reconfigurable Video Coding Framework." In: *Journal of Signal Processing Systems* 63.2 (May 2011), pp. 251–263. ISSN: 1939-8018, 1939-8115. DOI: 10.1007/s11265-009-0399-3.

[34] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications." In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. event-place: Toronto, Ontario, Canada. New York, NY, USA: Association for Computing Machinery, 2008, pp. 72–81. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128.

[35] Fischer Black and Myron Scholes. "The Pricing of Options and Corporate Liabilities." In: *Journal of Political Economy* 81.3 (1973). _eprint: https://doi.org/10.1086/260062, pp. 637–654. DOI: 10.1086/260062.

[36] Sebastian Blessing, Kiko Fernandez-Reyes, Albert Mingkun Yang, Sophia Drossopoulou, and Tobias Wrigstad. "Run, actor, run: towards cross-actor language benchmarking." In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2019*. the 9th ACM SIGPLAN International Workshop. Athens, Greece: ACM Press, 2019, pp. 41–50. ISBN: 978-1-4503-6982-4. DOI: 10.1145/3358499.3361224.

[37] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System." In: *Journal of Parallel and Distributed Computing* 37.1 (1996), pp. 55–69. ISSN: 0743-7315. DOI: https://doi.org/10.1006/jpdc.1996.0107.

[38] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. "Myrinet: a gigabit-per-second local area network." In: *IEEE Micro* 15.1 (1995), pp. 29–36.

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

'14. the Thirteenth ACM SIGPLAN workshop. Gothenburg, Sweden: ACM Press, 2014, pp. 13–23. ISBN: 978-1-4503-3038-1. DOI: 10.1145/2633448.2633453.

[40] Antonio Brogi, Andrea Canciani, Davide Neri, Luca Rinaldi, and Jacopo Soldani. "Towards a Reference Dataset of Microservice-Based Applications." In: *Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers*. Ed. by Antonio Cerone and Marco Roveri. Vol. 10729. Lecture Notes in Computer Science. Springer, 2017, pp. 219–229. DOI: 10.1007/978-3-319-74781-1_16.

[41] Antonio Brogi, Stefano Forti, Ahmad Ibrahim, and Luca Rinaldi. "Bonsai in the Fog: An active learning lab with Fog computing." In: *Third International Conference on Fog and Mobile Edge Computing, FMEC 2018, Barcelona, Spain, April 23-26, 2018*. IEEE, 2018, pp. 79–86. DOI: 10.1109/FMEC.2018.8364048.

[42] Antonio Brogi, Davide Neri, Luca Rinaldi, and Jacopo Soldani. "Orchestrating incomplete TOSCA applications with Docker." In: *Sci. Comput. Program.* 166 (2018), pp. 194–213. DOI: 10.1016/j.scico.2018.07.005.

[43] Antonio Brogi, Luca Rinaldi, and Jacopo Soldani. "TosKer: A synergy between TOSCA and Docker for orchestrating multicomponent applications." In: *Softw. Pract. Exp.* 48.11 (2018), pp. 2061–2079. DOI: 10.1002/spe.2625.

[44] Christopher Brown, Vladimir Janjic, Adam D. Barwell, J. Daniel Garcia, and Kenneth MacKenzie. "Refactoring GrPPI: Generic Refactoring for Generic Parallelism in C++." In: *International Journal of Parallel Programming* 48.4 (Aug. 2020), pp. 603–625. ISSN: 0885-7458, 1573-7640. DOI: 10.1007/s10766-020-00667-x.

[45] David R. Butenhof. *Programming with POSIX Threads*. 1997. ISBN: 0-201-63392-2.

[46] *CAF - C++ Actor Framework*. URL: http://actor-framework.org/ (visited on 06/03/2020).

[47] Rafael C. Cardoso, Maicon R. Zatelli, Jomi F. Hübner, and Rafael H. Bordini. "Towards benchmarking actor- and agent-based programming languages." In: *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control - AGERE! '13*. the 2013 workshop. Indianapolis, Indiana, USA: ACM Press, 2013, pp. 115–126. ISBN: 978-1-4503-2602-5. DOI: 10.1145/2541329.2541339.

[48] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. "FlumeJava: Easy, Efficient Data-Parallel Pipelines." In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. event-place: Toronto, Ontario, Canada. New York, NY, USA: Association for Computing Machinery, 2010, pp. 363–375. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806638.

[49] *Changing World, Changing Mozilla - The Mozilla Blog*. URL: https://blog.mozilla.org/blog/2020/08/11/changing-world-changing-mozilla/ (visited on 09/30/2020).

[50] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. "CAF - the C++ Actor Framework for Scalable and Resource-Efficient Applications." In: *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control - AGERE! '14*. the 4th International Workshop. Portland, Oregon, USA: ACM Press, 2014, pp. 15–28. ISBN: 978-1-4503-2189-1. DOI: 10.1145/2687357.2687363.

[51] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. "Revisiting actor programming in C++." In: *Computer Languages, Systems & Structures* 45 (Apr. 2016), pp. 105–131. ISSN: 14778424. DOI: 10.1016/j.cl.2016.01.002.

[52] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. "Native actors: a scalable software platform for distributed, heterogeneous environments." In: *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control - AGERE! '13*. the 2013 workshop. Indianapolis, Indiana, USA: ACM Press, 2013, pp. 87–96. ISBN: 978-1-4503-2602-5. DOI: 10.1145/2541329.2541336.

[53] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. "PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite." In: *ACM Trans. Archit. Code Optim.* 12.4 (Dec. 2015). Publisher: ACM, 41:1–41:22. ISSN: 1544-3566. DOI: 10.1145/2829952.

[54] Nianen Chen, Yue Yu, Shangping Ren, and Mattox Beckman. "A Role-Based Coordination Model and its Realization." In: *Informatica* 32.3 (2008), pp. 229–244.

[55] *Classic Routing • Akka Documentation*. URL: https://doc.akka.io/docs/akka/2.6.8/routing.html (visited on 06/16/2020).

[56] Sylvan Clebsch and Sophia Drossopoulou. "Fully concurrent garbage collection of actors on many-core machines." In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications -*

OOPSLA '13. the 2013 ACM SIGPLAN international conference. Indianapolis, Indiana, USA: ACM Press, 2013, pp. 553–570. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509557.

[57] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. "Deny capabilities for safe, fast actors." In: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE! 2015*. the 5th International Workshop. Pittsburgh, PA, USA: ACM Press, 2015, pp. 1–12. ISBN: 978-1-4503-3901-8. DOI: 10.1145/2824815.2824816.

[58] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine. "Glacier: Transitive Class Immutability for Java." In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 496–506. DOI: 10.1109/ICSE.2017.52.

[59] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press. Research Monographs in Parallel and Distributed Computing. Pitman, 1989. ISBN: 978-0-262-53086-6.

[60] Murray Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming." In: *Parallel Computing* 30.3 (2004), pp. 389–406. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2003.12.002.

[61] *Comedy: Node.js actor framework*. URL: https://github.com/untu/comedy (visited on 06/03/2020).

[62] *Contrasting Active Objects vs Tasks vs Actors – Carl Gibbs – Blog*. URL: http://www.carlgibbs.co.uk/blog/?p=237 (visited on 05/13/2020).

[63] *Cpp-rotor: Event loop friendly C++ actor micro-framework*. URL: https://github.com/basiliscos/cpp-rotor (visited on 06/03/2020).

[64] Silvia Crafa and Luca Tronchin. "Actors vs Shared Memory: two models at work on Big Data application frameworks." In: (2015). _eprint: 1505.03060, p. 19.

[65] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. "AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks." In: (2007), p. 10.

[66] *D Programming Language*. URL: https://dlang.org/ (visited on 06/03/2020).

[67] Emanuele D'Osualdo, Jonathan Kochems, and C. -H. Luke Ong. "Automatic Verification of Erlang-Style Concurrency." In: *Static Analysis*. Vol. 7935. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 454–476. ISBN: 978-3-642-38855-2. DOI: https://doi.org/10.1007/978-3-642-38856-9_24.

[68] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55.

[69] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured programming*. APIC studies in data processing. United States: Academic Press Inc., 1972. ISBN: 0-12-200550-3.

[70] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. "A divide-and-conquer parallel pattern implementation for multicores." In: *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems - SEPS 2016*. the 3rd International Workshop. Amsterdam, Netherlands: ACM Press, 2016, pp. 10–19. ISBN: 978-1-4503-4641-2. DOI: 10.1145/3002125.3002128.

[71] Marco Danelutto and Massimo Torquati. "Structured Parallel Programming with "core" FastFlow." In: *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*. Ed. by Viktória Zsók, Zoltán Horváth, and Lehel Csató. Cham: Springer International Publishing, 2015, pp. 29–75. ISBN: 978-3-319-15940-9. DOI: 10.1007/978-3-319-15940-9_2.

[72] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. "Parallel programming using skeleton functions." In: *PARLE '93 Parallel Architectures and Languages Europe*. Ed. by Arndt Bode, Mike Reeve, and Gottfried Wolf. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 146–160. ISBN: 978-3-540-47779-2.

[73] John Darlington, Yi-ke Guo, Hing Wing To, Jin Yang, Hing Wing, and To Jin Yang. "Parallel Skeletons for Structured Composition." In: ACM Press, 1995, pages.

[74] *Dart programming language*. URL: https://dart.dev/ (visited on 06/03/2020).

[75] Usman Dastgeer and Christoph Kessler. "Smart Containers and Skeleton Programming for GPU-Based Systems." In: *International Journal of Parallel Programming* 44.3 (June 1, 2016), pp. 506–530. ISSN: 1573-7640. DOI: 10.1007/s10766-015-0357-6.

[76] Usman Dastgeer, Lu Li, and Christoph Kessler. "Adaptive Implementation Selection in the SkePU Skeleton Programming Library." In: *Advanced Parallel Processing Technologies*. Ed. by Chenggang Wu and Albert Cohen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 170–183. ISBN: 978-3-642-45293-2.

[77] Adam L. Davis. "Akka Streams." In: *Reactive Streams in Java: Concurrency with RxJava, Reactor, and Akka Streams*. Berkeley, CA: Apress, 2019, pp. 57–70. ISBN: 978-1-4842-4176-9. DOI: 10.1007/978-1-4842-4176-9_6.

[78] Frank De Boer et al. "A Survey of Active Object Languages." In: *ACM Computing Surveys* 50.5 (Nov. 13, 2017), pp. 1–39. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3122848.

[79] Joeri De Koster, Stefan Marr, Theo D'Hondt, and Tom Van Cutsem. "Tanks: multiple reader, single writer actors." In: *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control - AGERE! '13*. the 2013 workshop. Indianapolis, Indiana, USA: ACM Press, 2013, pp. 61–68. ISBN: 978-1-4503-2602-5. DOI: 10.1145/2541329.2541331.

[80] Joeri De Koster, Stefan Marr, and Theo D'Hondt. "Synchronization Views for Event-loop Actors." In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. event-place: New Orleans, Louisiana, USA. New York, NY, USA: Association for Computing Machinery, 2012, pp. 317–318. ISBN: 978-1-4503-1160-1. DOI: 10.1145/2145816.2145873.

[81] Joeri De Koster, Stefan Marr, Tom Van Cutsem, and Theo D'Hondt. "Domains: Sharing state in the communicating event-loop actor model." In: *Computer Languages, Systems & Structures* 45 (Apr. 2016), pp. 132–160. ISSN: 14778424. DOI: 10.1016/j.cl.2016.01.003.

[82] Joeri De Koster, Tom Van Cutsem, and Theo D'Hondt. "Domains: safe sharing among actors." In: *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions - AGERE! '12*. the 2nd edition. Tucson, Arizona, USA: ACM Press, 2012, p. 11. ISBN: 978-1-4503-1630-9. DOI: 10.1145/2414639.2414644.

[83] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. "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*. the 6th International Workshop. Amsterdam, Netherlands: ACM Press, 2016, pp. 31–40. ISBN: 978-1-4503-4639-9. DOI: 10.1145/3001886.3001890.

[84] Tiziano De Matteis. "Parallel Patterns for Adaptive Data Stream Processing." PhD thesis. Pisa, Italy: University of Pisa, 2016. 184 pp.

[85] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. "Bringing Parallel Patterns Out of the Corner: The P^{3}ARSEC Benchmark Suite."

In: *ACM Trans. Archit. Code Optim.* 14.4 (Oct. 2017). Publisher: ACM, 33:1–33:26. ISSN: 1544-3566. DOI: 10.1145/3132710.

[86] Ugo de'Liguoro and Luca Padovani. "Mailbox Types for Unordered Interactions." In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Ed. by Todd Millstein. Vol. 109. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 15:1–15:28. ISBN: 978-3-95977-079-8. DOI: 10.4230/LIPIcs.ECOOP.2018.15.

[87] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1327452.1327492.

[88] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R Leblanc. "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions." In: *PROCEEDINGS OF THE IEEE* 87.4 (1999), p. 11.

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

[90] *E Programming Language*. URL: http://erights.org/ (visited on 06/03/2020).

[91] Kemal Ebciog, Vijay Saraswat, and Vivek Sarkar. "X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access." In: (2004), p. 11.

[92] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. "Statistical Model Checking for Composite Actor Systems." In: *Recent Trends in Algebraic Development Techniques*. Vol. 7841. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 143–160. ISBN: 978-3-642-37635-1. DOI: https://doi.org/10.1007/978-3-642-37635-1_9.

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

[94] *Elixir*. URL: https://elixir-lang.org/ (visited on 06/03/2020).

[95] Johan Enmyren and Christoph W. Kessler. "SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems." In: *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*. HLPP '10. event-place: Baltimore, Maryland, USA. New York, NY, USA: Association for Computing Machinery, 2010, pp. 5–14. ISBN: 978-1-4503-0254-8. DOI: 10.1145/1863482.1863487.

[96] *Erlang*. URL: https://www.erlang.org/ (visited on 06/03/2020).

[97] Steffen Ernsting and Herbert Kuchen. "Algorithmic skeletons for multi-core, multi-GPU systems and clusters." In: *International Journal of High Performance Computing and Networking* 7.2 (2012). Publisher: Inderscience Publishers Ltd, pp. 129–138.

[98] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. "SkePU 3: Portable High-Level Programming ofHeterogeneous Systems and HPC Clusters." In: 13th International Symposium onHigh-Level Parallel Programming and Applications (HLPP). Porto, Portugal, Sept. 7, 2020, pp. 18–37.

[99] August Ernstsson, Lu Li, and Christoph Kessler. "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems." In: *International Journal of Parallel Programming* 46.1 (Feb. 2018), pp. 62–80. ISSN: 0885-7458, 1573-7640. DOI: 10.1007/s10766-017-0490-5.

[100] *F Sharp MailboxProcessor*. URL: https://en.wikibooks.org/wiki/F_Sharp_Programming/MailboxProcessor (visited on 06/03/2020).

[101] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. "ParT: An Asynchronous Parallel Abstraction for Speculative Pipeline Computations." In: *Coordination Models and Languages*. Ed. by Alberto Lluch Lafuente and José Proença. Cham: Springer International Publishing, 2016, pp. 101–120. ISBN: 978-3-319-39519-7.

[102] M. J. Flynn. "Some Computer Organizations and Their Effectiveness." In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 1557-9956. DOI: 10.1109/TC.1972.5009071.

[103] Simon Fowler, Sam Lindley, and Philip Wadler. "Mixing Metaphors: Actors as Channels and Channels as Actors." In: (2017). In collab. with Marc Herbstritt. Artwork Size: 28 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 28 pages. DOI: 10.4230/LIPICS.ECOOP.2017.11.

[104] Emilio Francesquini, Alfredo Goldman, and Jean-François Méhaut. "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. event-place: Indianapolis, Indiana, USA. New York, NY, USA: Association for Computing Machinery, 2013, pp. 109–114. ISBN: 978-1-4503-2602-5. DOI: 10.1145/2541329.2541342.

[105] Emilio De Camargo Francesquini. "Dealing with actor runtime environments on hierarchical shared memory multi-core platforms." In: (2017), p. 216.

[106] J. Daniel Garcia, David del Rio, Marco Aldinucci, Fabio Tordini, Marco Danelutto, Gabriele Mencagli, and Massimo Torquati. "Challenging the abstraction penalty in parallel patterns libraries." In: *The Journal of Supercomputing* 76.7 (July 1, 2020), pp. 5139–5159. ISSN: 1573-0484. DOI: 10.1007/s11227-019-02826-5.

[107] Mauro Gaspari and Gianluigi Zavattaro. "An Algebra of Actors." In: *Formal Methods for Open Object-Based Distributed Systems*. Ed. by Paolo Ciancarini, Alessandro Fantechi, and Robert Gorrieri. Boston, MA: Springer US, 1999, pp. 3–18. ISBN: 978-0-387-35562-7.

[108] Leonardo Gazzarri and Marco Danelutto. "Supporting structured parallel program design, development and tuning in FastFlow." In: *The Journal of Supercomputing* 75.8 (Aug. 1, 2019), pp. 4026–4041. ISSN: 1573-0484. DOI: 10.1007/s11227-018-2448-9.

[109] B. Gedik, H. G. Özsema, and Ö Öztürk. "Pipelined fission for stream programs with dynamic selectivity and partitioned state." In: *Journal of Parallel and Distributed Computing* 96 (2016), pp. 106–120. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2016.05.003.

[110] Anastasia Gkolfi, Crystal Chang Din, Einar Broch Johnsen, Lars Michael Kristensen, Martin Steffen, and Ingrid Chieh Yu. "Translating active objects into colored Petri nets for communication analysis." In: *Science of Computer Programming* 181 (July 2019), pp. 1–26. ISSN: 01676423. DOI: 10.1016/j.scico.2019.04.002.

[111] Horacio González-Vélez and Mario Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers." In: *Software: Practice and Experience* 40.12 (Nov. 2010), pp. 1135–1160. ISSN: 00380644. DOI: 10.1002/spe.1026.

[112] M. L. Griss. "Software reuse: From library to factory." In: *IBM Systems Journal* 32.4 (1993), pp. 548–566.

[113] *Groups of Workers (experimental) — CAF 0.17.6 Documentation*. URL: https://actor-framework.readthedocs.io/en/0.17.6/ManagingGroupsOfWorkers.html (visited on 09/07/2020).

[114] Olivier Gruber and Fabienne Boyer. "Ownership-Based Isolation for Concurrent Actors on Multi-core Machines." In: *ECOOP 2013 – Object-Oriented Programming*. Vol. 7920. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 281–301.

ISBN: 978-3-642-39037-1. DOI: https://doi.org/10.1007/978-3-642-39038-8_12.

[115] Gaétan Hains, Ludovic Henrio, Pierre Leca, and Wijnand Suijlen. "Active Objects for Coordinating BSP Computations (Short Paper)." In: *Coordination Models and Languages*. Ed. by Giovanna Di Marzo Serugendo and Michele Loreti. Vol. 10852. Cham: Springer International Publishing, 2018, pp. 220–230. ISBN: 978-3-319-92407-6. DOI: 10.1007/978-3-319-92408-3_10.

[116] Tim Harris, James Larus, and Ravi Rajwar. "Transactional Memory, 2nd edition." In: *Synthesis Lectures on Computer Architecture* 5.1 (Dec. 22, 2010), pp. 1–263. ISSN: 1935-3235, 1935-3243. DOI: 10.2200/S00272ED1V01Y201006CAC011.

[117] Yaroslav Hayduk, Anita Sobe, and Pascal Felber. "Dynamic Message Processing and Transactional Memory in the Actor Model." In: *Distributed Applications and Interoperable Systems*. Vol. 9038. Cham: Springer International Publishing, 2015, pp. 94–107. ISBN: 978-3-319-19128-7. DOI: 10.1007/978-3-319-19129-4_8.

[118] Brandon Hedden and Xinghui Zhao. "A Comprehensive Study on Bugs in Actor Systems." In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018: 47th International Conference on Parallel Processing. Eugene OR USA: ACM, Aug. 13, 2018, pp. 1–9. ISBN: 978-1-4503-6510-9. DOI: 10.1145/3225058.3225139.

[119] Ludovic Henrio, Fabrice Huet, and Zsolt István. "Multi-threaded Active Objects." In: *Coordination Models and Languages*. Ed. by Rocco De Nicola and Christine Julien. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 90–104. ISBN: 978-3-642-38493-6.

[120] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures." In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA '93. event-place: San Diego, California, USA. New York, NY, USA: Association for Computing Machinery, 1993, pp. 289–300. ISBN: 0-8186-3810-9. DOI: 10.1145/165123.165164.

[121] Carl Hewitt and Henry Baker. "Laws for communicating parallel processes." In: *MIT Artificial Intelligence Laboratory Working Papers, WP-134A* (1977). Publisher: MIT Artificial Intelligence Laboratory.

[122] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence." In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. event-place: Stanford, USA. San Francisco,

CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[123] Raphael Hiesgen, Dominik Charousset, and Thomas C. Schmidt. "OpenCL Actors – Adding Data Parallelism to Actor-Based Programming with CAF." In: *Programming with Actors*. Ed. by Alessandro Ricci and Philipp Haller. Vol. 10789. Cham: Springer International Publishing, 2018, pp. 59–93. ISBN: 978-3-030-00301-2. DOI: 10.1007/978-3-030-00302-9_3.

[124] Jessica Hillert. "A Comparison of the Capability Systems of Encore, Pony and Rust." PhD thesis. 2019.

[125] Shams M. Imam and Vivek Sarkar. "Integrating Task Parallelism with Actors." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. event-place: Tucson, Arizona, USA. New York, NY, USA: Association for Computing Machinery, 2012, pp. 753–772. ISBN: 978-1-4503-1561-6. DOI: 10.1145/2384616.2384671.

[126] Shams M. Imam and Vivek Sarkar. "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*. the 4th International Workshop. Portland, Oregon, USA: ACM Press, 2014, pp. 67–80. ISBN: 978-1-4503-2189-1. DOI: 10.1145/2687357.2687368.

[127] Shams M. Imam and Vivek Sarkar. "Selectors: Actors with Multiple Guarded Mailboxes." In: *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control - AGERE! '14*. the 4th International Workshop. Portland, Oregon, USA: ACM Press, 2014, pp. 1–14. ISBN: 978-1-4503-2189-1. DOI: 10.1145/2687357.2687360.

[128] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in Dataflow Programming Languages." In: *ACM Comput. Surv.* 36.1 (Mar. 2004). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/1013208.1013209.

[129] Gilles Kahn. "The Semantics of a Simple Language for Parallel Programming." In: *IFIP Congress*. 1974.

[130] *karlrupp/microprocessor-trend-data: Data repository for my blog series on microprocessor trend data.* URL: https://github.com/karlrupp/microprocessor-trend-data (visited on 09/12/2020).

[131] *Kilim library*. URL: http://www.malhar.net/sriram/kilim/ (visited on 06/03/2020).

[132] Steve Klabnik and Carol Nichols. "Chapter 4: Understanding Ownership." In: *The Rust Programming Language*. no starch Press, 2018.

[133] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. no starch Press, 2018.

[134] Herbert Kuchen. "A Skeleton Library." In: *Euro-Par 2002 Parallel Processing*. Ed. by Burkhard Monien and Rainer Feldmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 620–629. ISBN: 978-3-540-45706-0.

[135] Herbert Kuchen and Murray Cole. "The integration of task and data parallel skeletons." In: *Parallel Processing Letters* 12.2 (2002). Publisher: World Scientific, pp. 141–155. DOI: 10.1142/S0129626402000896.

[136] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. "Twitter Heron: Stream Processing at Scale." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. event-place: Melbourne, Victoria, Australia. New York, NY, USA: Association for Computing Machinery, 2015, pp. 239–250. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742788.

[137] R. Greg Lavender and Douglas C. Schmidt. "Active Object: An Object Behavioral Pattern for Concurrent Programming." In: *Pattern Languages of Program Design 2*. Ed. by John M. Vlissides, James O. Coplien, and Norman L. Kerth. Section: Active Object: An Object Behavioral Pattern for Concurrent Programming. Addison-Wesley Longman Publishing Co., Inc., 1996, pp. 483–499. ISBN: 0-201-89527-7.

[138] *Laying the foundation for Rust's future | Rust Blog*. URL: https://blog.rust-lang.org/2020/08/18/laying-the-foundation-for-rusts-future.html (visited on 09/30/2020).

[139] Pierre Leca. "Combining active object and BSP programs." PhD thesis. Université Côte d'Azur, Feb. 10, 2020. 131 pp.

[140] Pierre Leca, Wijnand Suijlen, Ludovic Henrio, and Françoise Baude. "Distributed futures for efficient data transfer between parallel processes." In: *SAC 2020 - 35th ACM/SIGAPP Symposium On Applied Computing*. Brno, Czech Republic, Mar. 2020. DOI: 10.1145/3341105.3373932.

[141] Mario Leyton and Jose M Piquer. *Skandium: Multi-core Programming with Algorithmic Skeletons*. 2010. ISBN: 978-1-4244-5672-7.

[142] Henry Lieberman. *A Preview of Act 1*. URL: https://dspace.mit.edu/handle/1721.1/6350 (visited on 06/03/2020).

[143] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. "Ferret: A Toolkit for Content-Based Similarity Search of Feature-Rich Data." In: *SIGOPS Oper. Syst. Rev.* 40.4 (Apr. 2006). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 317–330. ISSN: 0163-5980. DOI: 10.1145/1218063.1217966.

[144] *Mailboxes • Akka Documentation*. URL: https://doc.akka.io/docs/akka/2.6.8/typed/mailboxes.html (visited on 06/16/2020).

[145] Sasikanth Manipatruni, Dmitri E. Nikonov, and Ian A. Young. "Material Targets for Scaling All-Spin Logic." In: *Phys. Rev. Applied* 5.1 (Jan. 2016). Publisher: American Physical Society, p. 014002. DOI: 10.1103/PhysRevApplied.5.014002.

[146] Nicholas D. Matsakis and Felix S. Klock II. "The Rust Language." In: *Ada Lett.* 34.3 (Oct. 2014). Place: New York, NY, USA Publisher: ACM, pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188.

[147] Kiminori Matsuzaki and Kento Emoto. "Lessons from Implementing the BiCGStab Method with SkeTo Library." In: *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*. HLPP '10. event-place: Baltimore, Maryland, USA. New York, NY, USA: Association for Computing Machinery, 2010, pp. 15–24. ISBN: 978-1-4503-0254-8. DOI: 10.1145/1863482.1863488.

[148] Timothy G Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Pearson Education, 2004. ISBN: 0-321-94078-4.

[149] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. Vol. 37. Place: New York, NY, USA Publisher: Association for Computing Machinery. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Nov. 2012. ISBN: 0-12-415993-1.

[150] Robert Ryan McCune, Tim Weninger, and Greg Madey. "Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing." In: *ACM Computing Surveys* 48.2 (Nov. 21, 2015), pp. 1–39. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2818185.

[151] Gabriele Mencagli, Massimo Torquati, Dalvan Griebler, Marco Danelutto, and Luiz Gustavo L. Fernandes. "Raising the Parallel Abstraction Level for Streaming Analytics Applications." In: *IEEE Access* 7 (2019), pp. 131944–131961. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2941183.

[152] G.E. Moore. "Cramming More Components Onto Integrated Circuits." In: *Proceedings of the IEEE* 86.1 (Jan. 1998), pp. 82–85. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.1998.658762.

[153] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications." In: *arXiv:1712.05889 [cs, stat]* (Dec. 15, 2017).

[154] *nact library*. URL: https://nact.io/ (visited on 06/03/2020).

[155] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. "Analytical Modeling of Pipeline Parallelism." In: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 2009, pp. 281–290.

[156] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. "Latency-Tolerant Software Distributed Shared Memory." In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '15. event-place: Santa Clara, CA. USA: USENIX Association, 2015, pp. 291–305. ISBN: 978-1-931971-22-5.

[157] *OpenCL Overview - The Khronos Group Inc*. URL: https://www.khronos.org/opencl/ (visited on 08/26/2020).

[158] *Orbit - Virtual actor framework*. URL: https://github.com/orbit/orbit/ (visited on 06/03/2020).

[159] *Orleans – Virtual Actors*. Sept. 14, 2014. URL: https://www.microsoft.com/en-us/research/project/orleans-virtual-actors/ (visited on 05/23/2020).

[160] *Orleans framework*. URL: https://dotnet.github.io/orleans/ (visited on 06/03/2020).

[161] Susanna Pelagatti. *Methodologies and tools for structured highly parallel computing*. PhD Thesis. University of Pisa, 1991.

[162] Susanna Pelagatti. *Structured development of parallel programs*. Vol. 102. Taylor & Francis Abington, 1998. ISBN: 0-7484-0759-6.

[163] F. Petrini, Wu-chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. "The Quadrics network: high-performance clustering technology." In: *IEEE Micro* 22.1 (2002), pp. 46–57.

[164] Greg Pfister. "An Introduction to the InfiniBand Architecture." In: *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. chapter 42. IEEE, 2002.

[165] *Pony*. URL: https://www.ponylang.io/ (visited on 06/03/2020).

[166] Alexander Pöppl, Scott Baden, and Michael Bader. "A UPC++ Actor Library and Its Evaluation On a Shallow Water Proxy Application." In: *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. 2019, pp. 11–24. DOI: 10.1109/PAW-ATM49560.2019.00007.

[167] Alexander Pöppl, Michael Bader, Tobias Schwarzer, and Michael Glaß. "SWE-X10: Simulating Shallow Water Waves with Lazy Activation of Patches Using Actorx10." In: *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*. 2016 Second International Workshop on Extreme-Scale Programming Models and Middleware (ESPM2). Salt Lake City, UT: IEEE, Nov. 2016, pp. 32–39. ISBN: 978-1-5090-3858-9. DOI: 10.1109/ESPM2.2016.010.

[168] Aleksandar Prokopec and Martin Odersky. "Isolates, channels, and event streams for composable distributed programming." In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) - Onward! 2015*. 2015 ACM International Symposium. Pittsburgh, PA, USA: ACM Press, 2015, pp. 171–182. ISBN: 978-1-4503-3688-8. DOI: 10.1145/2814228.2814245.

[169] *Proto.Actor Framework*. URL: http://proto.actor/ (visited on 06/03/2020).

[170] *Pykka library*. URL: https://www.pykka.org/en/latest/index.html (visited on 06/03/2020).

[171] *Reference Architectural Model for Industrie 4.0 (RAMI 4.0)*. URL: https://www.isa.org/intech-home/2019/march-april/features/rami-4-0-reference-architectural-model-for-industr (visited on 08/26/2020).

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

[173] *Riker Library*. URL: https://riker.rs/ (visited on 06/03/2020).

[174] Luca Rinaldi, Massimo Torquati, and Marco Danelutto. "Enforcing Reference Capability in FastFlow with Rust." In: *Parallel Computing: Technology Trends, Proceedings of the International Conference on Parallel Computing, PARCO 2019, Prague, Czech Republic, September 10-13, 2019*. Ed. by Ian T. Foster, Gerhard R. Joubert, Ludek Kucera, Wolfgang E. Nagel, and Frans J. Peters. Vol. 36. Advances in Parallel Computing. IOS Press, 2019, pp. 396–405. DOI: 10.3233/APC200064.

[175] Luca Rinaldi, Massimo Torquati, Daniele De Sensi, Gabriele Mencagli, and Marco Danelutto. "Are Actors Suited for HPC on Multi-Cores?" In: 12th International Symposium on High-Level Parallel Programming and Applications. Peer reviewed with internal proceedings. Linköping, Sweden, June 2019, p. 21.

[176] Luca Rinaldi, Massimo Torquati, Daniele De Sensi, Gabriele Mencagli, and Marco Danelutto. "Improving the Performance of Actors on Multi-cores with Parallel Patterns." In: *Interna-*

tional Journal of Parallel Programming (June 4, 2020). ISSN: 1573-7640. DOI: 10.1007/s10766-020-00663-1.

[177] Luca Rinaldi, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. "High-Throughput Stream Processing with Actors." In: *Proceedings of the 10th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2020. event-place: Virtual, USA. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–10. ISBN: 978-1-4503-8185-7. DOI: 10.1145/3427760.3428338.

[178] Luca Rinaldi, Massimo Torquati, Gabriele Mencagli, Marco Danelutto, and Tullio Menga. "Accelerating Actor-Based Applications with Parallel Patterns." In: *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). Pavia, Italy: IEEE, Feb. 2019, pp. 140–147. ISBN: 978-1-72811-644-0. DOI: 10.1109/EMPDP.2019.8671602.

[179] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. "A generic parallel pattern interface for stream and data processing." In: *Concurrency and Computation: Practice and Experience* 29.24 (2017). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4175, e4175. DOI: 10.1002/cpe.4175.

[180] Sascha Roloff, Alexander Pöppl, Tobias Schwarzer, Stefan Wildermann, Michael Bader, Michael Glaß, Frank Hannig, and Jürgen Teich. "ActorX10: an actor library for X10." In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10 - X10 2016*. the 6th ACM SIGPLAN Workshop. Santa Barbara, CA, USA: ACM Press, 2016, pp. 24–29. ISBN: 978-1-4503-4386-2. DOI: 10.1145/2931028.2931033.

[181] Thomas Rouvinez and Anita Sobe. "Comparison of Active Objects and the Actor Model." In: (2014), p. 9.

[182] *SALSA Programming Language*. URL: http://wcl.cs.rpi.edu/salsa/ (visited on 06/03/2020).

[183] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010. ISBN: 978-0-13-138768-3.

[184] *Scalaz Functional Library*. URL: https://scalaz.github.io/7/ (visited on 08/30/2020).

[185] Matthew Scarpino. "OpenCL in action: how to accelerate graphics and computations." In: (2011). Publisher: hgpu. org. ISSN: 978-1617290176.

[186] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. "Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model." In: *Science of Computer Programming* 80 (Feb. 2014), pp. 52–64. ISSN: 01676423. DOI: 10.1016/j.scico.2013.03.011.

[187] Amin Shali. "Actor Oriented Programming in Chapel." In: (2010), p. 16.

[188] shamsimam. *GitHub - shamsimam/savina: Savina is an Actor Benchmark Suite*. URL: https://github.com/shamsimam/savina (visited on 06/11/2020).

[189] Marjan Sirjani and Mohammad Mahdi Jaghoori. "Ten Years of Analyzing Actors: Rebeca Experience." In: *Formal Modeling: Actors, Open Systems, Biological Systems*. Vol. 7000. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 20–56. ISBN: 978-3-642-24932-7.

[190] David B. Skillicorn and Domenico Talia. "Models and Languages for Parallel Computation." In: *ACM COMPUTING SURVEYS* 30 (1998), pp. 123–169.

[191] *Slurm Workload Manager - Documentation*. URL: https://slurm.schedmd.com/ (visited on 09/21/2020).

[192] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI The Complete Reference, Volume 1: The MPI Core*. 2nd. (Revised). Cambridge, MA, USA: MIT Press, 1998. ISBN: 0-262-69215-5.

[193] sophialC. *GitHub - sophiaIC/pony-savina: Pony: Savina Benchmark Suite*. URL: https://github.com/sophiaIC/pony-savina (visited on 06/11/2020).

[194] *Streaming (experimental) — CAF 0.17.6 Documentation*. URL: https://actor-framework.readthedocs.io/en/0.17.6/Streaming.html (visited on 08/03/2020).

[195] Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. 2005. URL: http://www.gotw.ca/publications/concurrency-ddj.htm (visited on 12/27/2020).

[196] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[197] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. "Chocola: integrating futures, actors, and transactions." In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2018*. the 8th ACM SIGPLAN International Workshop. Boston, MA, USA: ACM Press, 2018, pp. 33–43. ISBN: 978-1-4503-6066-1. DOI: 10.1145/3281366.3281373.

[198] Janwillem Swalens, Stefan Marr, Joeri De Koster, and Tom Van Cutsem. "Towards Composable Concurrency Abstractions." In: *Electronic Proceedings in Theoretical Computer Science* 155 (June 12, 2014), pp. 54–60. ISSN: 2075-2180. DOI: 10.4204/EPTCS.155.8.

[199] *SYCL Overview - The Khronos Group Inc.* URL: https://www.khronos.org/sycl/ (visited on 10/28/2020).

[200] Carolyn L. Talcott. "An Actor Rewriting Theory." In: *Electronic Notes in Theoretical Computer Science* 4 (1996), pp. 361–384. ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)00047-7.

[201] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. "Why Do Scala Developers Mix the Actor Model with other Concurrency Models?" In: *ECOOP 2013 – Object-Oriented Programming*. Ed. by Giuseppe Castagna. Red. by David Hutchison et al. Vol. 7920. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 302–326. ISBN: 978-3-642-39037-1. DOI: 10.1007/978-3-642-39038-8_13.

[202] Rajeev Thakur, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, and Jesper Larsson Tr\{{\textbackslash}"a\}ff. "MPI at Exascale." In: *Procceedings of SciDAC*. Scientific Discovery through Advanced Computing (SciDAC). Vol. 2. 2010, pp. 14–35.

[203] *The chips are down for Moore's law : Nature News*. URL: https://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338 (visited on 09/12/2020).

[204] *The Encore Programming Language*. URL: https://stw.gitbooks.io/the-encore-programming-language/content/ (visited on 06/03/2020).

[205] *The LLVM Project. Clang: a C Language Family Frontend for LLVM*. URL: http://clang.llvm.org/ (visited on 09/04/2020).

[206] *The Scala Actors Library*. URL: https://docs.scala-lang.org/overviews/core/actors.html (visited on 06/03/2020).

[207] *Thespian Python Actors*. URL: https://thespianpy.com/doc/ (visited on 06/03/2020).

[208] Peter Thoman et al. "A taxonomy of task-based parallel programming technologies for high-performance computing." In: *The Journal of Supercomputing* 74.4 (Apr. 2018), pp. 1422–1434. ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-018-2238-4.

[209] Chris Tomlinson, Won Kim, Mark Scheevel, Vineet Singh, Becky Will, and Gul Agha. "Rosette: An object-oriented concurrent systems architecture." In: *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*. 1988, pp. 91–93.

[210] Massimo Torquati. "Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns." PhD thesis. Pisa, Italy: University of Pisa, Aug. 5, 2019.

[211] Massimo Torquati, Tullio Menga, Tiziano De Matteis, Daniele De Sensi, and Gabriele Mencagli. "Reducing Message Latency and CPU Utilization in the CAF Actor Framework." In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). Cambridge: IEEE, Mar. 2018, pp. 145–153. ISBN: 978-1-5386-4975-6. DOI: 10.1109/PDP2018.2018.00028.

[212] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. "A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs." In: *Programming with Actors*. Vol. 10789. Cham: Springer International Publishing, 2018, pp. 155–185. ISBN: 978-3-030-00301-2.

[213] Phil Trinder et al. "Scaling Reliably: Improving the Scalability of the Erlang Distributed Actor Platform." In: *ACM Transactions on Programming Languages and Systems* 39.4 (Aug. 17, 2017), pp. 1–46. ISSN: 01640925. DOI: 10.1145/3107937.

[214] Leslie G. Valiant. "A Bridging Model for Parallel Computation." In: *Commun. ACM* 33.8 (Aug. 1990). Place: New York, NY, USA Publisher: Association for Computing Machinery, pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181.

[215] Marco Vanneschi. *High performance computing: parallel processing models and architectures*. Pisa University Press, 2014. ISBN: 88-6741-372-4.

[216] Vaughn Vernon. *Reactive messaging patterns with the Actor model: applications and integration in Scala and Akka*. New York: Addison-Wesley, 2016. 448 pp. ISBN: 978-0-13-384683-6.

[217] Patrick Walton. *C++ design goals in the context of Rust*. URL: http://pcwalton.blogspot.com/2010/12/c-design-goals-in-context-of-rust.html (visited on 11/26/2020).

[218] Anthony Williams. *C++ Concurrency in Action*. Second Edition. Manning Publications, 2019. ISBN: 978-1-61729-469-3.

[219] woelke. *GitHub - woelke/savina CAF: Savina Benchmark Suite*. URL: https://github.com/woelke/savina/tree/caf_integration (visited on 06/11/2020).

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Final Version as of April 21, 2021 (`classicthesis` version 1.0).