Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200064

Enforcing Reference Capability in FastFlow with Rust

Luca Rinaldi [a,1], Massimo Torquati [a] and Marco Danelutto [a]
[a] *Computer Science Department, University of Pisa, Italy*

Abstract. In this work, we investigate the performance impact of using the `Rust` programming language instead of the `C++` one to implement two basic parallel patterns as provided by the FastFlow parallel library. The rationale of using `Rust` is that it is a modern system-level language capable to statically guarantee that if a data reference is sent over a communication channel, the ownership of the reference is transferred from the producer to the consumer. Such reference-passing semantics is at the base of the FastFlow programming model. However, the FastFlow library does not enforce nor checks its correct usage leaving this burden to the programmer. The results obtained comparing the FastFlow/`C++`, and the `Rust` implementations of the same implementation schema of the Task-Farm and Pipeline patterns show that `Rust` is a valid alternative to `C++` for the FastFlow library with indubitable benefits in terms of programmability.

Keywords. Multi-cores, parallel programming, reference capability, Rust, C++

1. Introduction

Multi-core and many-core processors are today largely used both in professional and consumer settings. Multi-cores are tightly-coupled Multiple-Instruction Multiple-Data (MIMD) architectures. They are shared-memory multiprocessors systems integrated into a single chip, often referred to as Chip Multi-Processors (CMP). Many-core processors are CMP systems that are designed to employ a high degree of parallelism (currently up to a few hundred cores), by using a large number of simpler cores than those used in general-purpose multi-cores. The broad diffusion of CMP systems has had and still is having, an important effect on how software is developed.

In these 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, therefore, increase the debugging and maintainability software phases.

A different approach is to use message passing semantics to coordinate the concurrent entities. A message induces an implicit synchronization between the sender and the receiver. This model may be used merely for synchronization purposes while data may

[1]Corresponding Author: Luca Rinaldi E-mail: luca.rinaldi@di.unipi.it

be shared exploiting the cache-coherent hardware capabilities of modern CMPs. Indeed, sharing mutable data in a producer-consumer fashion is generally more efficient than explicit copying, especially for large data structures. However data sharing is dangerous. Changes to a data reference might propagate producing unexpected data-races, i.e. two concurrent operations (where at least one is a write operation) to the same memory location without any synchronization.

The message passing model is used as a synchronization mechanism in some C++-based parallel library, such for example FastFlow [3] and GrPPI [5]. The `C++` programming language is commonly used for its large set of features and its performance. However, it does not provide strong guarantees for memory safety. Modern programming languages such as `Rust` [10] and Pony [4] have a *reference capability* system that statically checks access permissions to memory locations. In `Rust`, this feature is expressed with the concept of *ownership* [7]. The idea behind ownership is that, although multiple aliases to a resource may exist simultaneously, to perform specific actions on the resource (e.g., reading or writing a memory location) should require some unique capability *owned* by exactly one alias at any point in time during the execution of the program. This concept permits to enforce at compile time that every time a variable is sent over a communication channel, its ownership capability is also sent, so the sender cannot access the data anymore [11]. Such reference capability semantics is employed in the C++-based FastFlow parallel programming library. FastFlow is a library offering both high-level parallel patterns as well as composable parallel building blocks suitable for building run-time systems for new DSLs or for building new high-level parallel patterns. However, the reference capability semantics is not enforced by the FastFlow library, leaving the burden of respecting the semantics directly to the run-time system programmer.

In this work, we analyze the implications on the programming model and on the overall application performance of using the `Rust` language to implement the FastFlow parallel semantics. Specifically, we considered two simple synthetic benchmarks implemented by using two FastFlow parallel patterns: the Task-Farm pattern and the Pipeline pattern. These two patterns are particularly relevant because they are used in FastFlow as basic building blocks of other more complex parallel patterns (e.g., ParallelFor Divide&Conquer, and Macro Data-Flow). We aim to demonstrate that a system-level language such as `Rust` which provides strong statically checked features to the programmer can be a valid alternative to `C++` to write the parallel patterns offered by the FastFlow library. From the programming model standpoint, the `Rust` implementation has the additional advantage of statically enforcing the FastFlow producer-consumer semantics at the language level. To meet the objective, the FastFlow communication channel implemented as a Single-Producer Single-Consumer (SPSC) lock-free unbounded queue [2] has been adequately and safely wrapped to build a `Rust` library to be used for the implementation of inter-thread communication channels in `Rust`. The results obtained by running the synthetic benchmarks on a 24-core Intel multi-core platform, demonstrate that the `Rust` implementation of the benchmarks considered exhibits the same level of performance of the FastFlow `C++` implementation.

The remaining of this paper is organized as follows. The next section presents the background, specifically the FastFlow and the `Rust` language features. Then, Section 3 provides the motivations of this work. The experimental tests are described in Section 4. Finally, Section 5 briefly reviews similar works and summarizes our contributions.

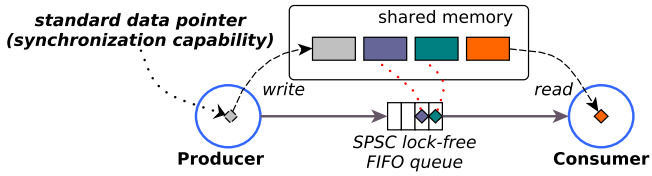

Figure 1. FastFlow library producer-consumer semantics: sending references to shared data over a SPSC lock-free FIFO channel.

2. Background

2.1. FastFlow

FastFlow is a C++ parallel programming library targeting multi/many-cores and offering a multi-level API to the parallel programmer [3,]. 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* and so on. At a lower level of abstraction, the library provides customizable sequential and parallel building blocks addressing the needs of the run-time system programmer. The idea is that new high-level patterns or new high-level libraries can be built by a proper assembly of the building blocks [1].

The library was conceived to support highly efficient stream parallel computations on heterogeneous multi-cores. The library is released open-source under the LGPLv3 licence [2].

The FastFlow library is realized as a modern C++ header-only template library that allows the programmer to simplify the development of parallel applications modeled 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 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-frequency streaming applications [2].

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) (see also the schema in Figure 1). The data reference is *de facto* a *capability*, i.e. a logical token that grants access to a given

[2] FastFlow home: http://calvados.di.unipi.it/fastflow

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.

2.2. `Rust`

`Rust` [10], is a modern system-level programming language that focuses on memory safety and performance.

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 [8]. 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 [8]: 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.

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 allocation 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.

`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. However, there are cases in witch the user has to annotate functions with life time parameters.

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. Almost every type in `Rust` implements `Send` and types composed entirely of types that are `Send`-able are themselves `Send`-able. The `Sync` trait indicates that it is safe for the type implementing `Sync` to be referenced from multiple threads.

3. Motivations

Many mainstream parallel programming libraries are written in C/`C++` primarily for performance reasons. Often, the burden of maintaining non-interference among threads implementing the application is in charge of the parallel programmer which has to correctly use either locks or (hopefully) suitable high-level parallel abstractions (e.g., parallel patterns). From the one hand, the usage of low-level synchronization mechanisms allows the programmer to have great flexibility and to tweak the code applying specific optimizations, but on the other hand, it exposes to potential unexpected behaviors and subtle data-races.

The so-called "modern C++" (i.e. C++11 and above) introduced *move semantics* and *smart pointers* features which greatly help the programmers to avoid errors related to pointer arithmetic without affecting (in the majority of cases) the overall performance. However, the responsibility to correctly use such new features is still in charge of the programmer that might not be a parallel programming expert. Moreover, in some situations, `C++` move semantics may produce additional data copies, for example in the one-to-many communication pattern implementing a data scattering operation.

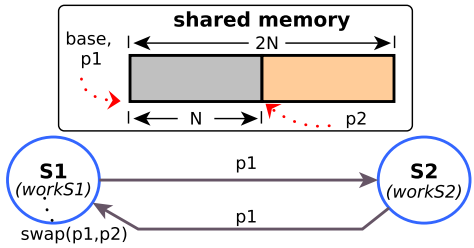

Figure 2. Logical schema of the FastFlow two-stage pipeline described in Listing 1.

As an example, a valid FastFlow program is the one sketched in Listing 1. 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` and then uses two raw pointers to point to two distinct parts of the vector that are swapped at every producer-consumer iteration. Each stage works on a portion of length `N` of the initial vector. The logical schema of this simple producer-consumer use-case is sketched in Figure 2.

In this simple example, there is no guarantee that within the `workS1` or `workS2` functions some wrong accesses to a portion of the vector may produce data-races due to buffer overruns. This kind of implementation would not be possible in the `Rust` language

```
struct Stage1: ff_node_t<float> {
  Stage1():base(2*N) {}
  int svc_init() {
    initialize(base);
    p1=base.data(); p2=p1+N;
    std::swap(p1,p2);
    return 0;
  }
  float* svc(float* in) {
    if(haveToStop(p1,p2))
       return EOS;
    std::swap(p1,p2);
    ff_send_out(p1);
    workS1(p2, N, 10.0);
    return GO_ON;
  }
  std::vector<float> base;
  float *p1,*p2;
} S1;
```

```
struct Stage2: ff_node_t<float> {
  float* svc(float* in) {
        workS2(in, N, 20.0);
        return in;
  }
} S2;

int main() {}
 // creates the pipeline
 ff_Pipe pipe(S1,S2);
 // creates the feedback channel
 pipe.wrap_around();
 // synchronous execution
 if(pipe.run_and_wait_end()<0) {
    error("running pipe\n");
    return -1;
 }
 return 0;
}
```

Listing 1: A simple producer consumer program in FastFlow

because the ownership rule is violated by the concurrent ownership of the vector by the two stages. In Rust, the programmer that wants to implement a similar program is forced to declare two separated vectors and to alternatively move the vectors' ownership through the communication channel connecting the two nodes. Moreover, accesses outside the boundaries of the two vectors is checked at run-time. It is worth noting that, a similar implementation is also possible in C++ but, while in Rust there is basically no other way to implement that program, in C++ there is nothing that may prevent a potentially dangerous implementation using raw pointers.

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 programming model. To alleviate the burden of the programmer, we decided to re-implement the FastFlow library using a language that can enforce reference capability at compile time. In this work, we want to measure the performance impact of using the Rust language with respect to a less-safe C++ implementation. Rust allows static checking at a higher level of abstraction than the one used to check the C++ move semantics. Our uphold that a proper combination of a system-level language with strong static checking features and a structured parallel programming methodology such the one offered by the FastFlow parallel library can significantly help the programmer to produce efficient and portable code with reduced programming effort and shorter time-to-solution.

4. Evaluation

In this section, we show the performance evaluation of using the Rust programming language instead of the C++ one in the implementation of two benchmarks based on two well-known FastFlow parallel patterns, namely the Task-Farm and the Pipeline. We selected these two patterns because they are used within the FastFlow library as basic building blocks for the implementation of other more complex parallel patterns.

4.1. Low-level mechanisms implementation

To have a fair performance comparison between the implementations of the two benchmarks, we need an implementation of the FastFlow communication channel in Rust. Initially, we considered to use the Multi-Producer Single-Consumer (MPSC) unbounded queue provided by the Rust standard library, but we found out that it does not deliver the expected performance, particularly for fine-grained computation. Therefore, we decided to port the C++-based FastFlow lock-free Single-Producer Single-Consumer (SPSC) unbounded queue [2] in Rust. However, instead of writing it from scratch mimicking the same FastFlow implementation, we decided to create a memory-safe Rust interface to the original C++-based FastFlow queue. The name of the Rust interface for the queue is `ff_buffer`[3].

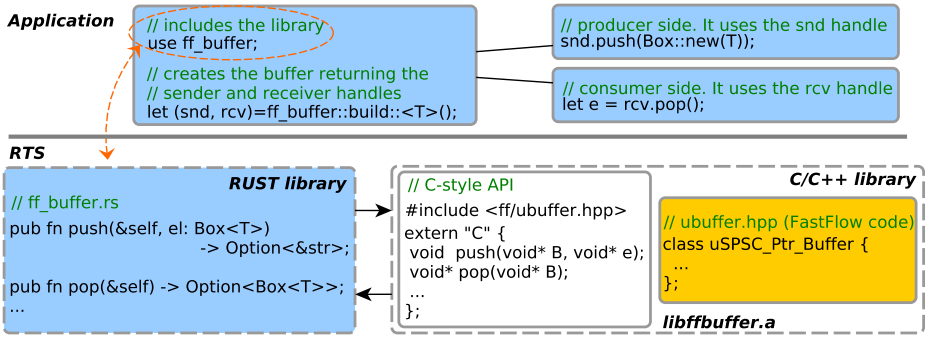

Figure 3. Integration of the FastFlow's unbounded SPSC lock-free queue in Rust.

Figure 3 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*[4] 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

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

cores. For this purpose we used the `Rust` third-party library `core_affinity`[5] to set the thread-to-core affinity for all `Rust` threads according to a simple round-robin assignment strategy.

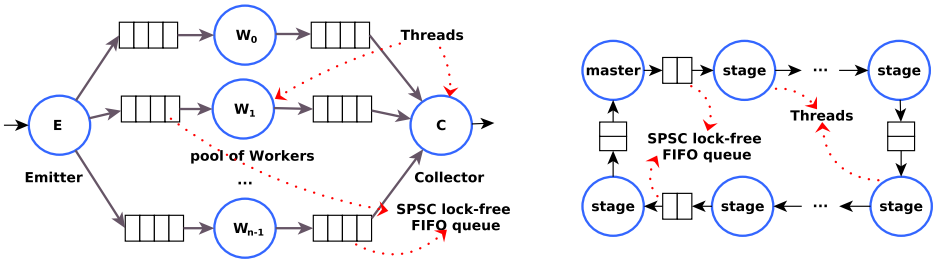

Figure 4. Implementation schema of the Task-Farm pattern (left-hand side) and of the Pipeline with feedback channel pattern (right-hand side).

In Figure 4 are 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. The one on the left-hand side 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. In the tests we executed, each Worker performs a configurable number of floating-point operations on each input data element. The Emitter node is in charge to assign 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 to study the scalability of the Task-Farm pattern by varying the number of Worker threads.

On the right-hand side of Figure 4 is shown 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.

4.2. Results

All tests reported in this section were conducted on an Intel Xeon Server equipped with two Intel E5-2695 Ivy Bridge CPUs running at 2.40GHz and featuring 24 cores (12 per socket). Each hyper-threaded core has 32KB private L1, 256KB private L2 and 30MB of L3 shared cache. The machine has 64GB of DDR3 RAM, running Linux 3.14.49 x86 64 with the CPUfreq performance governor enabled and turbo boost disabled. 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.

The tests were executed ten times, and the values reported in the plots is the average value of all runs. The standard deviation is small (less than 1%) and thus omitted for readability reasons.

[5] https://crates.io/crates/core_affinity

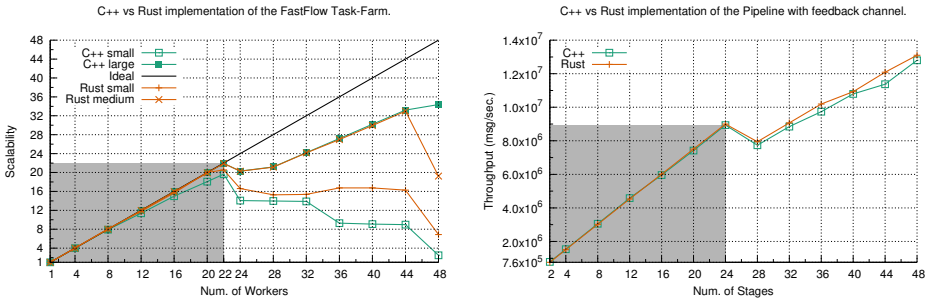

Figure 5. Left:) Scalability of the Task-Farm pattern implementation with two different computation granularities. Right:) Throughput of the Pipeline pattern with feedback channel varying the number of stages.

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). On the left-hand side of Figure 5 is shown the scalability of the Task-Farm pattern written in C++ (i.e. FASTFLOW v.3.0.0) 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 grey 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 on 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 5 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 that there is no significant performance difference between the `C++` and `Rust` versions for the two patterns considered.

5. Related Work and Summary

The `Rust` programming language is attracting increasing interest in the parallel community because of its comparable performance with `C/C++` and its memory safety.

Libraries such as *rsmpi*[6] and *Raycon*[7] are examples of well-known parallel programming libraries that moved from `C/C++` to `Rust`. Rsmpi is a MPI binding for `Rust`, and

[6] https://github.com/bsteinb/rsmpi
[7] https://github.com/rayon-rs/rayon

it permits to use the MPI library from within Rust programs. Raycon is a data parallel library similar to the OpenMP standard. It supports parallel computations such as *map*, *flap-map*, *filter*, *sorting* and *reduce* over Rust collections.

Other research works such as [6] and [9] try to improve and extend the Rust ownership system to better support parallel computations. The former proposes a statically checked communication protocol between threads. The latter proposes an extension of the ownership system where it is possible to specify that the same thread can own multiple times the same variable. Such extension simplifies code writing, especially in an event-based system, while maintaining the same security guarantees.

In this work, we evaluated the impact of statically enforcing the reference-passing semantics used in the FastFlow parallel programming library by using the Rust language features. We evaluated the impact on the performance of a Rust implementation of the Task-Farm and Pipeline pattern as provided by the FastFlow library. The results obtained show that the Rust language can be a valid alternative to the C++ one for implementing the FastFlow parallel patterns with several benefits in terms of programmability. However, more work is needed to build the entire software stack of the FastFlow library.

As future work, we intend to analyze and discuss the implementation of other parallel patterns and in particular of the Map one which poses non-trivial implementation problems if implemented in Rust.

References

[1] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Design patterns percolating to parallel programming framework implementation. *Int. J. Parallel Program.*, 42(6):1012–1031, 2014.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi-core systems. In C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 662–673, Berlin, Heidelberg, 2012. Springer.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: High-Level and Efficient Streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. John Wiley & Sons, Inc, Jan. 2017.

[4] S. Clebsch, S. Drossopoulou, S. Blessing, and A. 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, pages 1–12, New York, NY, USA, 2015. ACM.

[5] D. del Rio Astorga, M. F. Dolz, J. Fernandez, and J. D. Garca. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, pages e4175–n/a, 2017. e4175 cpe.4175.

[6] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 13–22. ACM, 2015.

[7] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, Dec. 2017.

[8] S. Klabnik and C. Nichols. *Chapter 4: Understanding Ownership*. no starch Press, 2018.

[9] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, pages 21–26. ACM, 2015.

[10] N. D. Matsakis and F. S. Klock, II. The rust language. *Ada Lett.*, 34(3):103–104, Oct. 2014.

[11] Z. Yu, L. Song, and Y. Zhang. Fearless concurrency? understanding concurrent programming safety in real-world rust software. *arXiv preprint arXiv:1902.01906*, 2019.