Exploring stream parallel patterns in distributed MPI environments

Javier López-Gómez, Javier Fernández Muñoz, David del Rio Astorga, Manuel F. Dolz, J. Daniel García

University Carlos III of Madrid, Spain
{jalopezg,jfmunoz}@inf.uc3m.es, drio@pa.uc3m.es, dolzm@icc.uji.es, jdgarcia@inf.uc3m.es

Abstract

In recent years, the large volumes of stream data and the near real-time requirements of data streaming applications have exacerbated the need for new scalable algorithms and programming interfaces for distributed and shared-memory platforms. To contribute in this direction, this paper presents a new distributed MPI back end for GrPPI*, a C++ high-level generic interface of data-intensive and stream processing parallel patterns. This back end, as a new execution policy, supports distributed and hybrid (distributed+shared-memory) parallel executions of the* Pipeline *and* Farm *patterns, where the hybrid mode combines the MPI policy with a* GrPPI *shared-memory one. These patterns internally leverage distributed queues, which can be configured to use two-sided or one-sided MPI primitives to communicate items among nodes. A detailed analysis of the* GrPPI *MPI execution policy reports considerable benefits from the programmability, flexibility and readability points of view. The experimental evaluation of two different streaming applications with different distributed and shared-memory scenarios reports considerable performance gains with respect to the sequential versions at the expense of negligible* GrPPI *overheads.*

Keywords Parallel Patterns, Stream Processing, Distributed Patterns, C++ Programming, Generic Programming

Formal publication `https://doi.org/10.1016/j.parco.2019.03.004`

I. Introduction

Numerous scientific experiments, ranging from particle accelerators to environmental sensors, are nowadays generating large volumes of streaming data which has to be processed in near real-time. To enable the next generation of scientific discoveries, several challenges have been identified in the high performance computing (HPC) domain to handle the high throughput rates and low latency demands of data stream processing applications (DaSP) [1]. These challenges, envisioned to fill the gap in managing and processing streaming data, focus on the research of scalable and efficient streaming algorithms, programming models, languages and runtime systems for this type of applications. For that, considerable efforts in the HPC software stack for DaSP are required to face such incessant growth of streaming data [2].

To achieve the scalability goal, multiple shared-memory multi-core platforms are required to increase performance in this kind of applications. In this sense, the *de facto* programming models for distributed- and shared-memory platforms are, respectively, the MPI [3] and OpenMP [4] interfaces, which can be used in conjunction to enable hybrid parallelism. Regardless of their efficiency, both interfaces offer low-level abstractions and demand considerable expertise on the application and system domains to fine-tune the target application [5]. An opportunity to relieve this burden is to use pattern-based programming models, which encapsulate algorithmic aspects following a building blocks approach. Generally, parallel patterns offer an alternative to implement robust, readable and portable solutions, hiding away complexities related to concurrency mechanisms, synchronizations or data sharing [6]. In this respect, some pattern-based interfaces and libraries from the state-of-the-art, e.g. FastFlow [7], Muesli [8] or SkePU 2 [9], already support clusters of multi-core machines and make use of scheduling algorithms to improve load balancing among nodes.

To pave the way towards HPC scalable programming models for DaSP, in this paper we extend our C++ generic and reusable parallel pattern interface (GrPPI) [10] with a new MPI back end, which enables the execution of some streaming patterns on distributed platforms. Basically, GrPPI accommodates a unified layer of generic and reusable parallel patterns on top of existing execution environments and pattern-based frameworks. This layer allows users to make their applications independent of the parallel programming framework used underneath, thus providing portable and readable codes. With this first version of the MPI back end, we support the distributed and hybrid execution of the Pipeline and Farm stream patterns. To support hybrid scenarios, the back end combines an intra-node shared-memory execution policy that if needed, is used to run multiple Pipeline or Farm operators inside an MPI process using OpenMP, C++ Threads or Intel TBB. Specifically, this paper contributed to the following:

- We present the new GrPPI MPI execution policy for distributed and hybrid environments for both Pipeline and Farm parallel patterns.
- We describe the design of the GrPPI interface and the MPI policy internals to allow the distributed and hybrid execution of DaSP applications.
- We present the distributed multiple-producer/multiple-consumer queues for the GrPPI stream operators. These queues allow the transferring of data items using two-sided (send/receive) and one-sided (get/put/fetch-and-op/ compare-and-swap) MPI primitives.
- We evaluate the distributed Pipeline and Farm stream patterns from the usability and performance points of view using two

streaming applications: *i)* a video processing applications and *ii)* an application that renders Mandelbrot frames. This evaluation is carried out under different hybrid configurations.

- We analyze the pattern usability in terms of lines of code and cyclomatic complexity, and perform a side-by-side comparison of both GrPPI and MPI programming interfaces.

In general, this paper extends the results presented in [11] with *i)* the implementation of the communication channels using one-side MPI primitives; *ii)* the extended evaluation using a video processing application; and *iii)* the analysis and comparison of both distributed queue communication models (two-sided and one-sided) under the different application scenarios.

The rest of this paper is organized as follows. Section II revisits some related works in the area. Section III reviews some basics about the stream patterns targeted in this paper and describes the GrPPI interface in detail. Section IV describes in detail the MPI queues with its two communication nodes: two-sided and one-sided. Section V describes the user interface of the proposed MPI back end, along with the algorithm that distributes operators among MPI processes. Section VI evaluates the performance of a stream-processing application under different distributed and hybrid configurations and the interface usability. Finally, Section VII closes the paper with a few concluding remarks and future works.

II. Related work

In the literature, a considerable collection of research works can be found about data stream processing in scientific HPC applications targeted to distributed platforms. We classify these works in the following two categories: *i)* stream processing engines and frameworks; and *ii)* pattern-based programming environments for distributed systems.

Regarding the first category, we identify some popular stream processing engines such as Storm [12], Spark [13] and Flink [14] targeted to clusters and cloud environments. Basically, these engines offer APIs that allow programmers to implement their applications as directed flow graphs, the nodes being operators and the edges stream flows. Depending on how these operators and flows are arranged, different and complex operations for splitting/joining the streams, and filtering data can be performed. While these stream engines are mainly Big Data and IoT-oriented, we also find HPC-oriented stream programming frameworks. A key example is StreamIt [15], a language that enables high performance of large streaming applications by efficiently mapping processes to a wide range of environments, including shared-memory architectures and HPC clusters. Another example is MPIStream, a prototype library implemented atop MPI, which provides an interface to existing MPI applications to adopt the streaming model [16, 17]. Basically, MPIStream provides a lightweight approach to link MPI processes with different tasks by using four basic concepts of stream processing: communication channels, data producer/consumer, data streams and stream operations.

On the other hand, we encounter pattern-based parallel programming frameworks tackling distributed systems. For instance, the authors of Fastflow in [18] report an extension of this library targeting cluster multi-core workstations, where the ZeroMQ library is used as the external asynchronous communication layer. Another example is Eden [19], an extension providing patterns in Haskell which gives support for parallel and distributed environments. In this library, processes communicate through unidirectional channels that are defined by programmers while specifying data dependencies. Similarly, JaSkel [20] provides sequential, concurrent, and distributed versions of the pipeline and farm skeletons, being possible to deploy them on both cluster and grid infrastructures.

Alternatively, we also find parallel-pattern programming frameworks which make use of MPI for targeting distributed platforms. An example is SkeTo [21], a C++ library coupled with MPI that offers operations for parallel data structures such as lists, trees, and matrices, however, it lacks stream-oriented patterns. In a similar way, the Muesli skeleton library [22] offers a large collection of patterns through C++ methods implemented in OpenMP and MPI, for multi-core and cluster platforms, respectively. The major supported patterns are distributed arrays and matrices for data parallelism; and pipelines and farms for stream-oriented parallelism. Another contribution is MALLBA [23], a library that provides a collection of high-level skeletons for combinatorial optimization which deals with parallelism in a user-friendly and efficient manner. MALLBA leverages NetStream, a custom MPI abstraction layer that takes care of primitive data type marshaling and synchronization between processes running in distributed machines. Finally, we highlight DASH [24], a C++ template library that offers distributed data structures and parallel Standard template library (STL) algorithms via a compiler-free Partitioned Global Address Space (PGAS) approach.

Given the foregoing, we identify an important gap between the MPI community and stream processing needs of today's scientific applications [1]. In this sense, the objective of the new GrPPI MPI back end is twofold. On the one hand, GrPPI offers a high-level C++ interface of parallel patterns that improves both application flexibility and source code readability [10]. On the other hand, the use of GrPPI MPI back end transparently enables the execution of streaming C++ application on HPC distributed platforms, as GrPPI hides away the complexity related to communication and process synchronization.

III. Background

In this section, we describe the necessary background about the Pipeline and Farm streaming parallel patterns tackled in this paper and review the GrPPI parallel pattern interface, where the new distributed MPI back end has been implemented.

III.1 Streaming parallel patterns

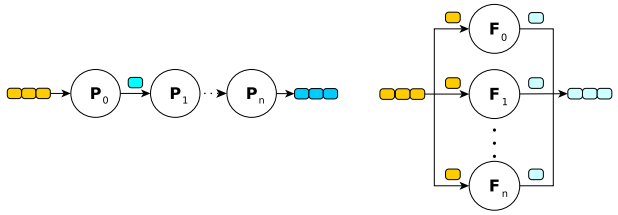
In general, DaSP applications can be seen as data-flows in the form of directed acyclic graphs (DAG), where the root nodes (*producers*) receive items from some given input stream, intermediate nodes perform some kind of operation on them, and leaf nodes (*consumers*) dump processed items onto an output stream. To accelerate these applications, the nodes, or operators, can be executed in parallel as long as data item dependencies are preserved. A common and simple DAG construction in DaSP is the Pipeline pattern, where the operators in a topological sort have data item dependencies only with the previous operator. Another common construction is the Farm pattern, where an operator is replicated n times to increase its

throughput, so multiple stream items can be computed in parallel. The formal definitions of the Pipeline and Farm patterns are the following:

Pipeline This pattern processes the items appearing on the input stream in several parallel stages. Each stage of this pattern processes data produced by the previous stage in the pipe and delivers results to the next one. Provided that the i-th stage in a n-staged Pipeline computes the function $f_i : \alpha \rightarrow \beta$, the Pipeline delivers the item x_i to the output stream applying the function $f_n(f_{n-1}(\ldots f_1(x_i)\ldots))$. The main requirement of this pattern is that the functions related to the stages should be independent among them, i.e., they can be computed in parallel without side effects. The parallel implementation of this pattern is performed using a set of concurrent entities, each of them taking care of a single stage. Figure 1(a) shows the Pipeline diagram.

Farm This pattern computes in parallel the function $f : \alpha \rightarrow \beta$ over all the items appearing in the input stream. Thus, for each item x_i on the input stream the Farm pattern delivers an item to the output stream as $f(x_i)$. In this pattern, the computations performed by the function for the input stream items should be completely independent of each other, otherwise, it cannot be processed in parallel. Thus, the function f can be computed in parallel by the different concurrent entities. Figure 1(b) depicts the Farm diagram.

As stated, these patterns can be composed among them to produce more efficient applications. Basically, the compositions supported between the Pipeline and Farm patterns are those in which the Pipeline stages can be parallelized individually using the Farm pattern. Thus, if a Pipeline stage corresponds with a pure function, this can be computed in parallel following a Farm construction. Throughout this paper, we denote the sequential stages of a Pipeline with "p", the Farm stages with "f" and the communication between two stages with the symbol "|". For instance, a Pipeline comprised of 4 stages, where the second and the third are Farm stages, is represented by "(p|f|f|p)".



(a) Pipeline pattern. (b) Farm pattern.

Figure 1: Pipeline and Farm pattern diagrams.

III.2 GrPPI, a generic parallel pattern interface

Regarding the parallel pattern interface, we have leveraged GrPPI, a generic and reusable parallel pattern interface for C++ applications [10]. This interface takes full advantage of modern C++ features, metaprogramming concepts, and generic programming to act as a switch between the OpenMP, C++ threads and Intel TBB parallel programming models. Its design allows users to leverage the aforementioned execution frameworks in a single and compact interface, hiding away the complexity behind the use of concurrency mechanisms. Furthermore, the modularity of GrPPI permits to easily integrate new patterns, while combining them to arrange more complex constructions. Also, it allows the integration of new execution policies based on distributed and shared-memory programming models. Thanks to this properties, GrPPI can be used to implement a wide range of existing stream-processing and data-intensive applications with relatively small efforts, having as a result portable codes that can be executed on multiple platforms. Figure 2 depicts the general view of the GrPPI library.

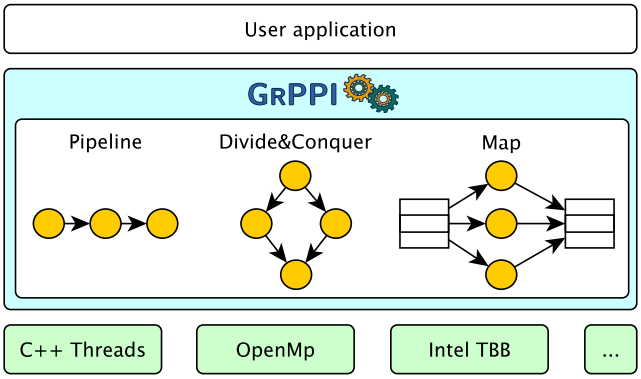


Figure 2: GrPPI architecture.

Listing 3 shows both Pipeline and Farm GrPPI C++ pattern prototypes. Note the use of `templates` with simple and variadic (parameter pack) parameters as universal references, allowing to pass any callable object (e.g., a functor or lambda expression) for the pattern operators. Note as well that the first parameter indicates the execution policy that shall be used to execute the operators.

```
template <typename E, typename G, typename ... O>
void pipeline(E execution_policy, G && generator, O &&
     ... operators);
```

(a) Pipeline pattern.

```
template <typename O>
void farm(int num_replicas, O && operator);
```

(b) Farm pattern.

Figure 3: Pipeline and Farm GrPPI pattern interfaces.

IV. MPI communication queues

Communication channels are fundamental in distributed DaSP applications. For instance, a stream operator running on a node needs to receive items from the previous operator mapped on another node, perform some computation on them, and send them to the next operator. Thus, to enable stream parallelism across nodes in our new MPI

execution policy, we determine the need for communication channels in the form of First-In-First-Out (FIFO) queues. For that, given that MPI does not natively support distributed queues, we propose two different implementations using one-sided and two-sided MPI communications primitives. Basically, these queues encapsulate the communications among MPI processes running stream operators. This way, the pattern implementations only make use of the public queue functions, e.g., the MPI queue producer side calls `push` in order to enqueue an item which should be sent to the next operator; while the consumer side calls to `pop` so as to dequeue an item by receiving it from the previous operator.

Considering the possible arrangements between the Pipeline and Farm GrPPI patterns, we identify four different communication scenarios:

- *Single-Producer, Single-Consumer* (SPSC): two consecutive Pipeline stages (`p|p`).
- *Single-Producer, Multiple-Consumer* (SPMC): a Pipeline stage to a set of Farm replicas (`p|f`).
- *Multiple-Producer, Single-Consumer* (MPSC): a set of Farm replicas to a Pipeline stage (`f|p`).
- *Multiple-Producer, Multiple-Consumer* (MPMC): a set Farm replicas to another set of Farm replicas (`f|f`).

Internally, these queues call to the corresponding MPI primitives depending on the selected communication mode; for two-sided we use `MPI_Send` and `MPI_Recv`, while for one-sided we leverage `MPI_Get` and `MPI_Put`. In the following sections, we explain in detail the implementation of these communication modes in our MPI distributed queues.

IV.1 Two-sided communication queues

As mentioned, the two-sided communication mode is implemented on top of `send` and `receive` MPI primitives, using the MPI interface provided by the Boost MPI library [25]. To illustrate the behaviour of this queue, left-hand side of Figures 4(a) and 4(b) show, respectively, the queue diagrams in the SPSC and MPMC scenarios.[1] As shown in the MPMC queue producer side, the queue creates a controller thread per producer process ($T_{0,1}$) in order to overlap communications with the operator computations. The queue also selects a controller thread that acts as the orchestrator, which is responsible for managing the queue by keeping track of the items available in each producer and forwarding consumer item requests to a given producer. For the SPSC queue, however, since the producer and consumer processes are known in advance, the communications can be overlapped with computations by directly using MPI asynchronous send/receive primitives. Therefore, no controller threads in the producer side are needed to manage the queue when dealing with SPSC scenarios.

To implement the aforementioned behavior, both producer and consumer MPI queue sides communicate following a specific protocol depending on the queue scenario (see right-hand side Figures 4(a) and 4(b) for SPSC and MPMC, respectively). This communication protocol is comprised of the three following phases:

- **Configuration phase**: The goal of this phase is to determine whether the queue should be run in SPSC or MPMC mode. If there exist only one producer and one consumer, then the queue is configured as SPSC; otherwise, it is set as MPMC. At the beginning of this phase, both producers and consumers initialize their queue sides according to the operator type (Pipeline stage or Farm replica). From this point on, all processes exchange different configuration messages with the orchestrator process so as to select the queue mode. First, the consumers and producers send the respective messages `reg_recv<id, type>` and `reg_send<id>` for registering themselves to the orchestrator. Additionally, consumers indicate the orchestrator their `type`, i.e. Pipeline stage or Farm replica. Next, the orchestrator configures the queue as SPSC or MPMC depending on the number of registered consumers and producers, and communicate them the acknowledge queue scenario (`ack<mode>`). Finally, producer processes launch the corresponding controller threads if the queue is configured as MPMC.

- **Communication phase**: A different communication protocol is used depending on the queue mode. If the queue is configured as SPSC, the producer process asynchronously sends the items to the consumer. Otherwise, if the queue is set to MPMC mode, the orchestrator thread waits for messages coming from producers and consumers. Producers send `notify_item<id>` messages to inform about a new available item, while consumers send `item_req<id>` to ask the orchestrator for an item. When the orchestrator receives a new request, it serves the item directly or forwards the request to another producer, following the same order in which the producer notifications arrived.

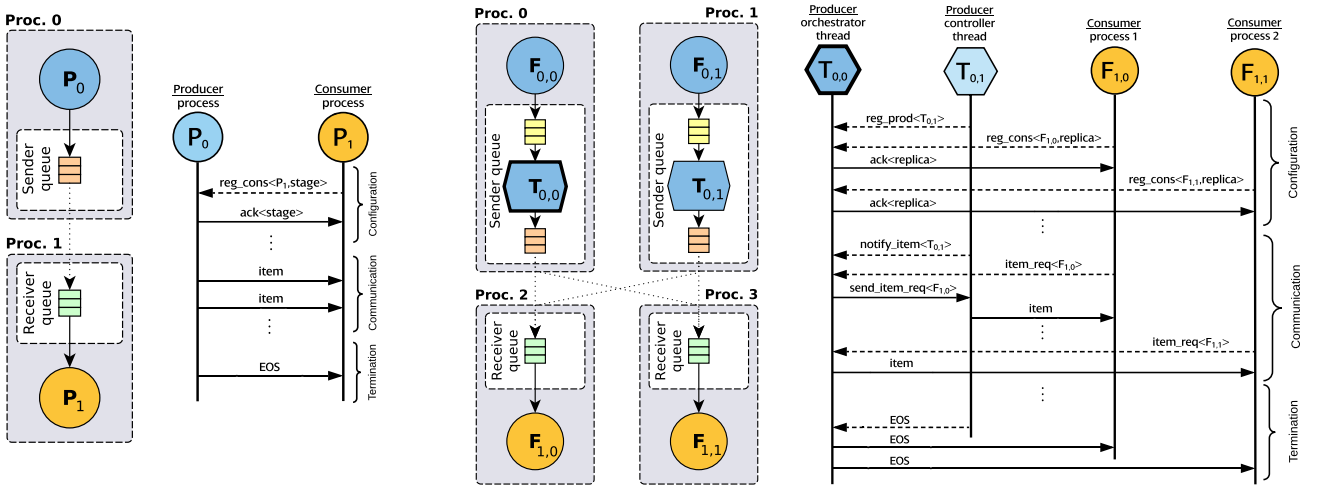
- **Termination phase**: Similar to the previous phase, a different finalization protocol is used depending on the queue mode. For the SPSC mode, as soon as the producer finishes its operation, it sends the *end-of-stream* (`EOS`) message to the consumer. On the contrary, for the MPMC mode, the orchestrator waits for `EOS` messages from all producers, including its own termination item. When this happens, the orchestrator sends to each of the consumers the `EOS` message to finalize the communication.

Thanks to these queues, stream operators run by MPI processes are able to transmit items according to the stream flow dictated by a concrete Pipeline construction, which can be composed of different Farm patterns. Note that if a process uses multiple queue instances simultaneously, MPI tags, instead of multiple communicators, are used to reference each of them.

IV.2 One-sided communication queues

To support the one-sided communication mode we leverage both `MPI_Get` and `MPI_Put` primitives. This type of communication was defined in the MPI 2 standard by introducing Remote Memory Access (RMA), also called one-sided communications because they require only one process to transfer data [26]. One-sided communication allows a process (origin) to read/write memory of a target process. For physical layers that support RDMA, e.g. Infiniband, the target process does not need to intervene during the communication. This fact benefits our distributed queue as a consumer is able to fetch a new item directly from the memory of a producer. Note that,

[1] We consider the SPMC and MPSC scenarios are implemented using the MPMC queue mode, as these cases can be seen as specializations of MPMC.



(a) SPSC queue diagram and communication protocol.

(b) MPMC queue diagram and communication protocol.

Figure 4: SPSC and MPMC two-side queues for the MPI execution policy.

in this case, we employ the MPI C API since the Boost MPI library does not provide support for one-sided operations.

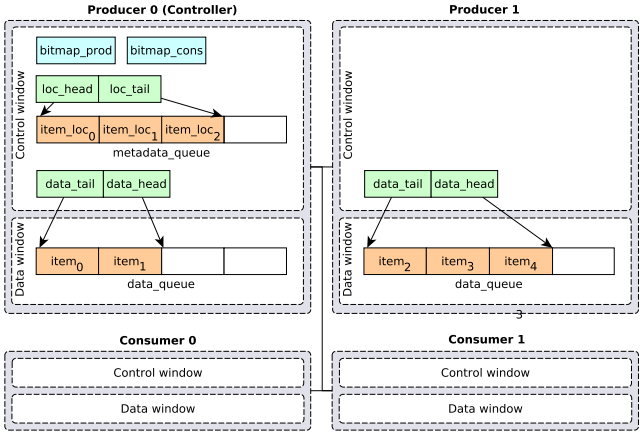


Figure 5: MPMC one-side queue schema.

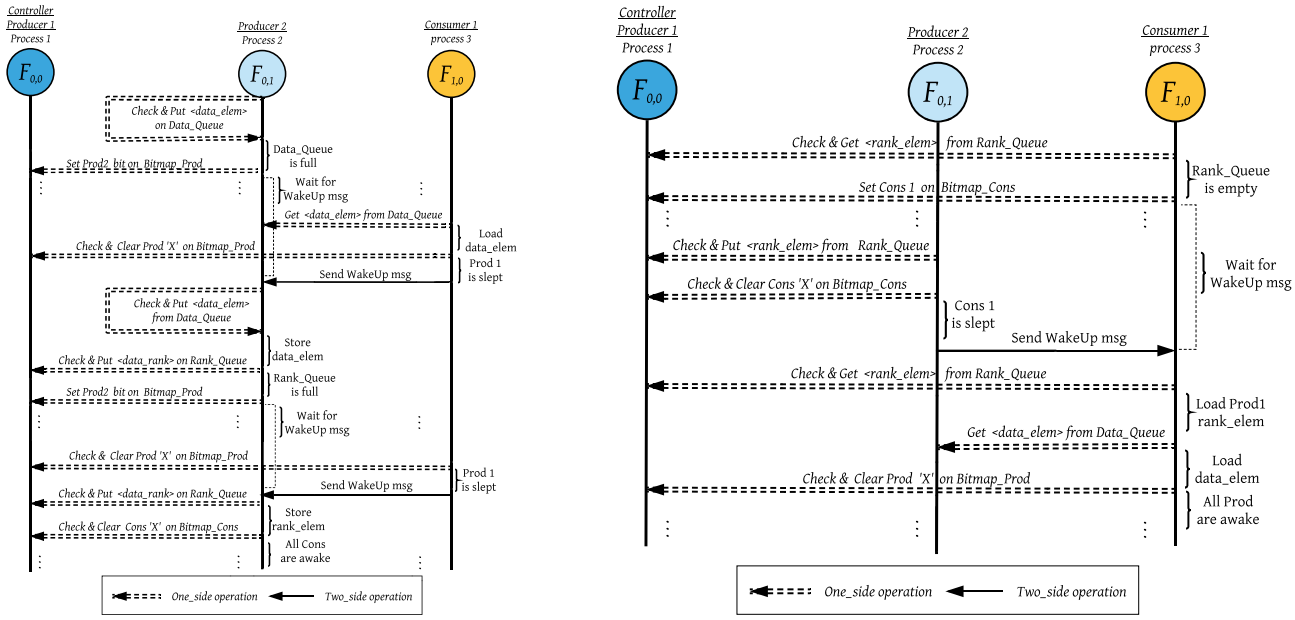
To communicate data items among stream operators, we leverage MPI windows for two different purposes: data and control. Figure 5 depicts the general schema of a one-sided communication queue and the use of such windows. As can be seen, producers use data windows to store the serialized data (items) which are transmitted through the nodes executing consecutive stream operators. Alternatively, control windows are used to keep track of the data window head and tail pointers. Further, the process acting as the queue controller stores a circular list of $(rank, offset, length)$ tuples to describe available data in one of the producers. This way, each time a data item becomes available in a producer, it is annotated in the list. When a consumer requests a new element, it retrieves the first tuple in the list and reads *length* bytes, starting at *offset* from the data window of the process with rank *rank*.

Control windows also store two bitmaps which are used to annotate the waiting producers and consumers that cannot introduce or consume an element because their queue is already full or empty. In these cases, when a producer or a consumer cannot push/pop an element from the queue, it is annotated as a waiting process in the bitmap, blocking the process on a `receive` primitive. Afterwards, when an item is consumed (or new data becomes available), the consumer (or producer) will send a wake-up message to one of the waiting processes to notify that the queue is ready. This technique avoids unnecessary communications to repeatedly check if new items can be consumed or produced.

To enable the one-sided model, we have defined a protocol allowing to instantiate queues and create the necessary memory windows. This communication protocol is comprised of the three following phases:

- **Initialization phase**: The initialization stage in charge of generating the corresponding communicators for the processes involved in sending/receiving items to/from a queue. In this stage, each of the processes constructs a bitmap in which each bit indicates whether a particular queue is referenced. A queue is referenced if the process uses it either for producing (sending) or consuming (receiving) items. Next, these bitmaps are shared among all the processes using the `MPI_Allgather` primitive. Afterwards, all the processes involved in a queue call the `MPI_Comm_create_group` function providing a group generated using the received bitmaps. Finally, the processes involved in this new communicator initialize their respective windows using the collective communication primitive `MPI_Win_create`.

- **Communication phase**: Once the queue is initialized, it can start communicating elements from/to multiple producers/consumers. Figure 6 illustrates two examples of communication from both the producer and consumer side. The first step for a producer to store a new data item is to check whether the data



(a) Producer-side communication protocol.

(b) Consumer-side communication protocol.

Figure 6: MPMC one-side queues for the MPI execution policy.

window has some space left. If the window is full, the producer sets the bit corresponding to its rank in the controller (producer-wait bitmap) and waits until receiving a message indicating that a consumer obtained an item from its data window. Afterwards, the producer may try to introduce a new data item in its internal data queue. If the queue is not full, the producer stores the data item in such queue and tries to introduce the tuple corresponding to the location of the item in the metadata queue (at the controller). Similarly to introducing the item in the data queue, the producer checks whether the metadata queue is full. This way, after introducing the location in the metadata queue, the item becomes available for the consumer processes. Finally, the producer checks the consumer-wait bitmap and sends a wake up message to one of the waiting consumers, if any.

With regards to the consumer process (see Figure 6(b)), the first step is to obtain an item location from the metadata queue. When the queue is empty, the consumer sets its corresponding bit in the consumer bitmap (stored in the controller) and waits for a wake up message from a producer. After receiving the message, the consumer process tries to obtain an item location. If the consumer is able to obtain a data location from the queue, it will try to read the corresponding data from the data window of the process that stores the item. Finally, this process selects a waiting producer to send a wake-up message.

- **Finalization phase**: Finally, to control the end of the stream, the controller process exposes an atomic counter of producers that have not generated the end-of-stream item yet. A producer that reaches the end-of-stream decrements this counter, and sends the end-of-stream message to all of the consumers if there are no other producers left. After all the consumers receive this message, the queue object may be destroyed. As part of the destructor, RMA windows are deallocated using the `MPI_Win_free` primitive. `MPI_Win_free` is a collective call that acts as a barrier, so producers memory will not be released until all the involved processes call the queue destructor.

In general, the one-sided MPI queue mode avoids the use of the send/receive primitives, as the consumers have access to the memory window exposed by the producers where data items reside. Further, the one-sided queue mode avoids the use of the orchestrator thread used in the two-sided mode. However, to leverage the one-sided communications potential, a network with RDMA support is required. Also, for multi-threaded stream operators, the memory window cannot be opened/accessed by multiple threads simultaneously, turning hybrid (distributed + shared-memory) scenarios more difficult to implement and requiring the use of synchronization primitives (locks). This fact, however, does not occur in the two-sided queue mode, where the use of the thread safety support (`MPI_THREAD_MULTIPLE`) combined with tags and group communicators eases the internal implementation of the distributed MPMC queues and the programmability of multi-threaded stream operators.

V. The GrPPI MPI execution policy

As previously stated, the goal of GrPPI is to accommodate a layer of parallel patterns between developers and existing parallel programming frameworks. So far, GrPPI only supports shared-memory

execution policies, such as C++ Threads, OpenMP and Intel TBB. To extend GrPPI in order to support distributed platforms, we have incorporated the MPI execution policy which, at this moment, allows users to execute the Pipeline and Farm stream patterns on multi-core clusters. In this section, we describe in detail the basic elements of the MPI execution policy: the user interface and mapping algorithm.

V.1 User interface

Given the GrPPI design, where each execution policy is implemented using a C++ class, for the new MPI policy we also designed its class. This is because each of these classes contains the framework-specific pattern implementations along with the configuration parameters for that policy. Basically, GrPPI pattern interfaces are overloaded with a different implementation for each of the available execution policies. With it, when the user code is compiled, the specific pattern implementation is selected depending on the execution policy passed as argument (see the `execution_policy` parameter in the Pipeline interface in Figure 3). Listing 1 shows an excerpt of the MPI execution policy class.

As observed, the execution policy constructors receive the program arguments directly. Additionally, the second argument determine the preferred communication mode using an enumerate that can take the values `one_sided` or `two_sided`. To support hybrid scenarios, both constructors can additionally receive a local shared-memory execution policy. This local policy will be used to run multiple Pipeline stages/Farm replicas (also referred to as stream operators) inside the same process. Therefore, operators assigned to the same process will be executed in sequential or in parallel depending on the selected shared-memory execution policy.

Listing 1: GrPPI MPI execution policy class.

```
template<typename LocalPolicy = parallel_execution_native>
class parallel_execution_mpi {
 ...
 public:
  parallel_execution_mpi(int argc, char** argv,
    CommMode comm_mode = one_sided,
    LocalPolicy local_exec_policy =
         parallel_execution_native{});
  parallel_execution_mpi(
    CommMode comm_mode = one_sided,
    LocalPolicy local_exec_policy =
         parallel_execution_native{});
  ...
};
```

As shown in Listing 2, the GrPPI Pipeline pattern leverages the new MPI execution policy class for its execution. Note that the policy is constructed at the declaration using both `argc` and `argv` arguments. This Pipeline is comprised of three stages in the form `(p|f|p)`, where the first and third stages run in series, while the second executes two replicas of the same operator using the Farm pattern. Assuming that the program is run by 4 MPI processes, the first and last respectively execute the generator and consumer stages, while the second and third will compute each of the Farm replicas. It is important to remark that items transiting from one stage to the next are sent and received through the MPI communication queues implemented within the policy back end.

Listing 2: Example of GrPPI distributed Pipeline.

```
grppi::parallel_execution_mpi ex{argc, argv, grppi::
     two_sided};
grppi::pipeline(ex,
  [x=1,n]() mutable -> optional<double> {
    if (x<=n) return x++;
    else return {};
  },
  grppi::farm(2,[](double x) {
    return x*x;
  }),
  [](double x) { cout << x << endl; }
);
```

V.2 Mapping stream operators onto processes

The MPI execution policy also embeds a mapping algorithm to assign stream operators onto MPI processes. Basically, this algorithm calculates at the beginning of the Pipeline execution the total number of operators involved in it, considering both the number of stages and Farm replicas. Afterwards, it computes the number of operators that should be run per MPI process (opp). By default, the operators are distributed homogeneously using the formula:

$$opp = \frac{num\_ops}{num\_procs} \tag{1}$$

However, the user can override this value by calling the function `set_grouping_granularity(int ops_per_proc)`, part of the MPI execution policy class. Next, each of the MPI processes calculates, using its rank, the range of operators that should be executed with the following formula:

$$Range(rank) = \begin{cases} \{rank * opp, ((rank+1) * opp) - 1\} & \text{if } rank \neq num\_procs - 1 \\ \{rank * opp, num\_ops - 1\} & \text{if } rank = num\_procs - 1 \end{cases} \tag{2}$$

According to this formula, if the opp value is set by default i.e. with Eq. 1, all processes execute the same number of operators except the last, which executes all remaining operators. Otherwise, if opp has been set by the user and is bigger than that set by default, then some of the last processes might not execute any operator. Note as well that the mapping algorithm selects consecutive operators to be mapped onto MPI processes, i.e. following the same order as they appear in the Pipeline pattern.

Figure 7 depicts a four-stage Pipeline composed of two Farm patterns in the second and third stages, running with 5 replicas each. In this case, the grouping granularity or opp has been computed by default using Eq. 1. Given that, each process executes the 3 consecutive operators corresponding to those returned by Eq. 2. Thus, the same MPI process can execute, using the local shared-memory policy, both Pipeline stages and/or Farm replicas.

All in all, this MPI execution policy allows GrPPI patterns to be executed on distributed multi-core platforms, exploiting both inter and intra-node parallelism. Also, thanks to its operator mapping algorithm, it is able to automatically distribute operators following the logical streaming order.

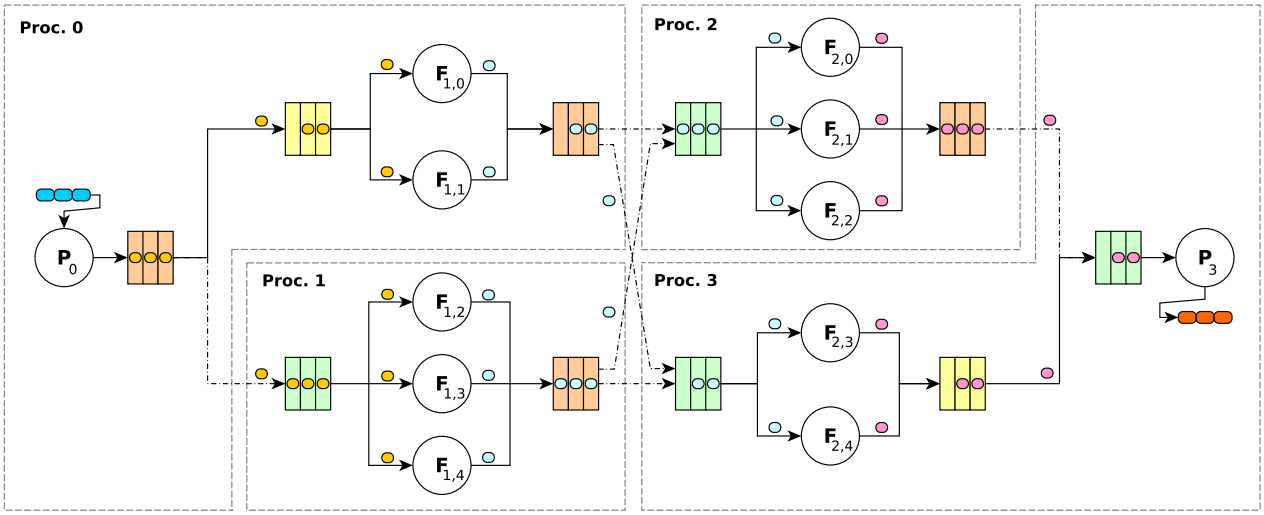


Figure 7: Distribution of stream operators onto processes with $opp = 3$.

VI. Experimental evaluation

In this section, we perform an experimental evaluation of the GrPPI MPI back end from the performance and interface productivity points of view. For this evaluation, we employ the following hardware and software components:

- **Target platform**. The evaluation has been carried out on two different cluster platforms:

  Tucan This platform is a homogeneous eight-node cluster, each node comprising 2× Intel Xeon Harpertown E5405 with 4 cores (total of 8 cores) running at 2.00 GHz, 12 MB of L2 cache and 8 GB of DDR3 RAM. The OS is a Linux Ubuntu 16.04.3 LTS with the kernel 4.4.0-97. Nodes are interconnected using a 1 Gigabit Ethernet switch.

  Tintorrum This platform is another homogeneous eight-node cluster, where each node is equipped with an Intel Xeon Westmere E5645 comprised of 6 cores running at 2.40 GHz, 12 MB of L3 cache and 48 GB of DDR3 RAM. The OS is CentOS 6.6 with the kernel 2.6.32-504. The nodes in this cluster are connected through an InfiniBand QDR network using Mellanox MTS3600 switch.

- **Software**. We leveraged the new GrPPI MPI back end built on top of GrPPI v0.4 [27], along with the respective shared-memory and distributed-memory back ends, C++11 threads and MPI-3.1, implemented by OpenMPI 3.1.2. Note that the MPI back end was implemented using the Boost MPI v1.66.0. The C++ compiler used to assemble GrPPI was GCC v6.3.0 which already supports the C++14 standard.

- **Use cases**. To evaluate both the Pipeline and Farm distributed patterns, we leverage two different use cases:

  Video-App The video processing use case is a synthetic streaming application in charge of applying a Gaussian blur and a Sobel filter over a set of images and generating an output video file with the resulting images. The pipeline of this application is composed of the four stages: *i)* a generator, which returns the file names of the different images to be processed; *ii)* a Gaussian blur filter; *iii)* a Sobel operator; and *iv)* a consumer, which turns each to the processed images to an output video frame. Note that the blur and Sobel filter, i.e., second and third stages of the pipeline, can be executed in parallel using the Farm pattern. This is because the images are completely independent of each other.

  Concretely, this synthetic benchmark will be used to evaluate both distributed Pipeline and Farm patterns under different configurations of kernel size for the Gaussian blur and Sobel operators and number of processes/threads per process on the Farm patterns computing both blur and Sobel filters.

  Mandelbrot The Mandelbrot use case is a streaming application that computes Mandelbrot set images for building a fractal zoom animation and applying the Gaussian blur filter on each of the generated frames. Concretely, this Pipeline-based application consists of the following stages: *i)* a Generator, which returns monotonically linear increasing zoom values; *ii)* a Mandelbrot stage, receiving zoom values for computing the Mandelbrot frame corresponding to such zoom value; *iii)* a Gaussian blur filter, receiving the frames from the previous stage and applying the Gaussian blur filter using a 3×3 pixels kernel; *iv)* a consumer, which dumps each of the frames onto the disk using the BMP format. In this use case, since the animation frames can be computed independently, both Mandelbrot and Gaussian blur Pipeline stages can be replicated by means of the Farm pattern, so they can process individual frames. Thus, several compositions may arise depending on the parallelization of these stages, i.e., `(p|p|p|p)`, `(p|f|p|p)`, `(p|p|f|p)`, and `(p|f|f|p)`.

As is generally known, the Mandelbrot set is the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, etc., remains bounded in absolute value. Thus, Mandelbrot images may be created by sampling the complex numbers and determining, for each sample point c, whether the result of iterating the above function goes to infinity. Afterwards, treating the real and imaginary parts of c as image coordinates $(x + yi)$ on the complex plane, pixels may then be colored according to how rapidly the sequence $z_n^2 + c$ diverges. This is key to understand the heterogeneous nature of this operation, where the divergence speed at the point of interest and zoom value dictate the number of iterations to compute a given frame. On the contrary, the Gaussian blur operator has an almost homogeneous workload to process each of the frames.

In the following sections, we analyze the performance and the productivity of the GRPPI distributed Pipeline and Farm patterns using the above-mentioned benchmarks with varying configurations of parallelism degree w.r.t. the number of MPI processes and worker threads used in Farm stages. Also, we evaluate the performance of both MPI queue communication modes, i.e. two-sided and one-sided. Finally, we evaluate the productivity of the GRPPI interface on the MANDELBROT use case by means of analyzing the number of lines of code (LOCs) and the Cyclomatic Complexity Number (CCN) required to implement the application with and without GRPPI.

VI.1 Performance analysis of Video-App

In this section, we evaluate the performance of VIDEO-APP with using images of size 400×400 pixels on both TUCAN and TINTORRUM platforms. Figure 8 shows the speedup obtained for both MPI queue communication modes w.r.t. the number of threads per node on TUCAN. It is important to remark that each experiment runs as many operators (Pipeline stages and Farm replicas) as the total number of threads executed across nodes. As can be observed, both MPI queue communication modes (Figures 8(a) and 8(b)) scale w.r.t. the number of nodes and threads used. Since MPI one-sided communications over Ethernet are implemented by means of TCP/IP segments, both one-sided and two-sided versions perform similarly. Additionally, the speedup attained by MPI using a single node is equal to the shared memory back end; this is because of the MPI back end delegates completely on the shared memory back end when running on one node.

Figure 9 shows the results for the same experiment on the TINTORRUM platform using the InfiniBand network. As it can be noticed, the efficiency obtained for both MPI queue communication modes and for all number of threads per core is higher than the efficiency attained on TUCAN. This improvement is given by the bisection network bandwidth of InfiniBand over Ethernet. It can also be clearly seen that the one-sided communication mode outperforms the two-sided mode; this is because the RDMA capability is natively supported by the InfiniBand interconnect technology. This capability prevents the processes sharing memory windows from being directly involved in the communications, as it RDMA is handled at the hardware level.

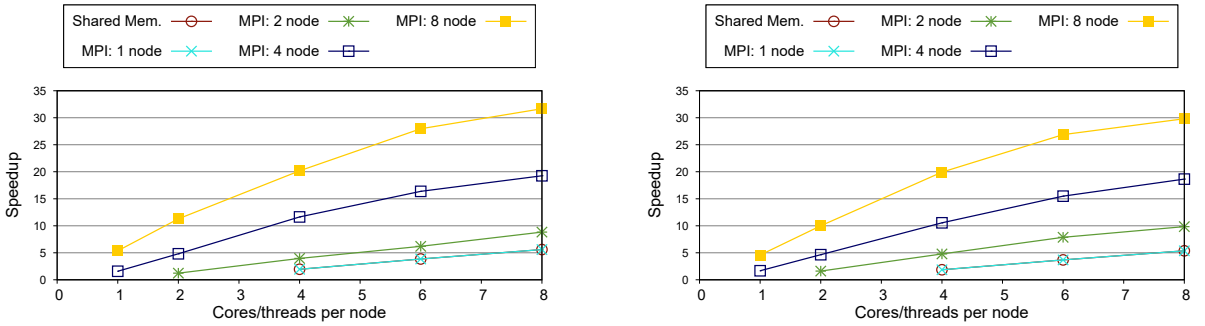
VI.1.1 Performance analysis of Mandelbrot

In this section, we evaluate the performance and the scalability of the MANDELBROT application implemented with the GRPPI interface using the MPI policy along with the shared-memory back end based on C++11 threads. Basically, we analyze the behavior of the two-sided communication MPI back end using equal and adjusted number of replicas on the Farm operators. Next, we analyze the performance on TUCAN and TINTORRUM platforms using both one-sided and two-sided communication modes with the best configuration of the number of replicas from the previous experiment.

Figure 10 depicts the speedup scaling when using from 1 to 8 TUCAN nodes and executing from 1 to 8 threads per node for square frame resolutions 400 w.r.t. the sequential application. In this first experiment attempt (see Figure 10(a)), we set the same number of Farm replicas in both Mandelbrot and Gaussian blur stages. As can be seen, for 1, 2 and 4 nodes, the application linearly scales with the number of threads per node, having a sustained efficiency of roughly 47 %. However, for 8 nodes the performance scaling degrades from 6 threads per node on since the Mandelbrot stage causes a major bottleneck in the Pipeline due to unbalanced stage throughputs. The maximum efficiency, in this case, is 35 %. This is because the Mandelbrot workload per frame is much higher than applying the Gaussian blur operator.

To deal with this unbalance, we have empirically calculated the ratio between the Mandelbrot and blur stages throughputs, which served us to determine the optimal number of replicas in their corresponding Farm stages. Basically, we have calculated this ratio using average throughputs of both stages for the different frame resolutions. Using this ratio, we assign 1 blur replica per each 25 Mandelbrot replicas. Nevertheless, this ratio does not deliver ideal throughputs given the heterogeneous nature of the Mandelbrot set workload. Figure 10(b) shows the performance achieved for the different number of nodes and threads per node experiments and using the aforementioned ratio. As observed, from 1 to 4 nodes, the efficiency significantly improves w.r.t. the previous experiment, from 47 % to 80 %. On the other hand, for 8 nodes, the application shows worse performance when using more than 6 threads per core. This degradation is mainly due to the difference between the stage workload and the inherent communication overheads. For this reason, for large frame sizes the performance attained is slightly better, as the parallel computations pay off the required data serialization and transfer times. In any case, for 8 nodes, the efficiency obtained using balanced Farm stages increases from 35 % to 70 %.

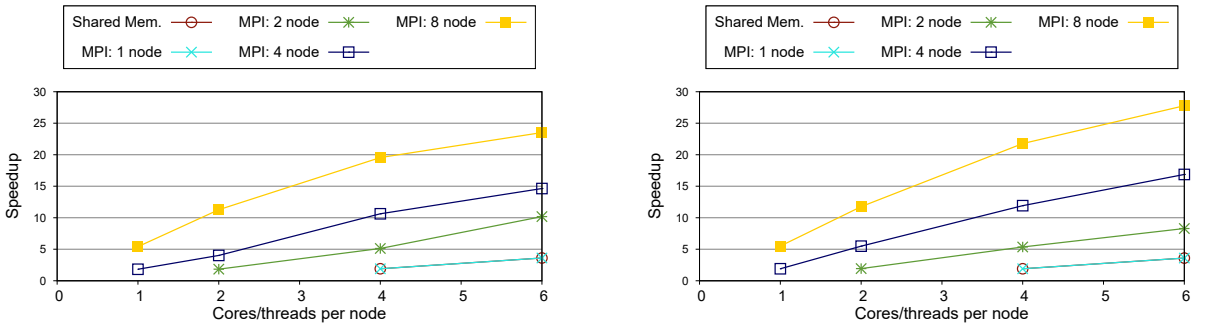
To extend our evaluation, we now compare the performance of the balanced Farm stages using both MPI communication modes: two-sided and one-sided on TUCAN and TINTORRUM platforms. Figure 11 shows the speedup scaling on TUCAN using the aforementioned communication modes. As observed, the performance obtained on both two-sided and one-sided communication modes is quite similar for all number of nodes, scaling mostly linearly with increasing number of threads per node. Contrarily, the same experiment performed on TINTORRUM, as shown in Figure 12 reveals that using one-sided communications improve the speedup scaling. As stated in the previous section, this is due to the RDMA capability of InfiniBand and the higher bisection bandwidth of the TINTORRUM network. Therefore, this kind of streaming communication-intensive applications



(a) Speedup using two-sided communications.

(b) Speedup using one-sided communications.

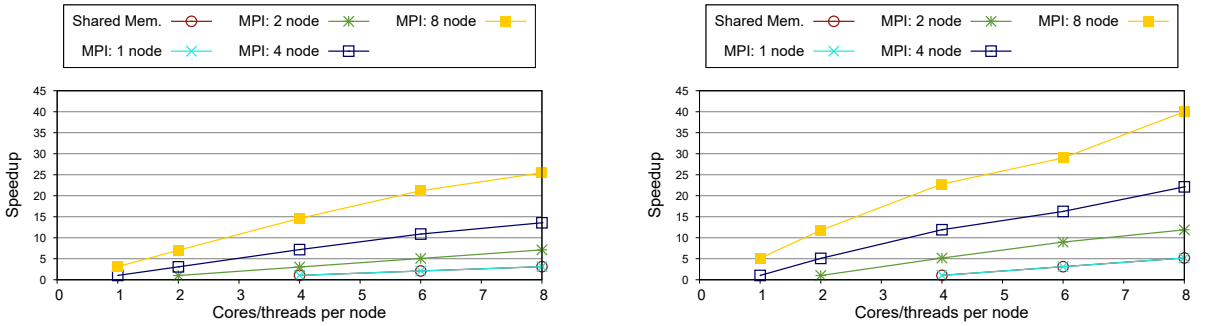
Figure 8: Speedup of the image use case using images of size 400×400 *on* Tucan.



(a) Speedup using two-sided communications.

(b) Speedup using one-sided communications.

Figure 9: Speedup of the image use case using images of size 400×400 *on* Tintorrum.



(a) Speedup using equal number of replicas per Farm.

(b) Speedup using adjusted number of replicas per Farm.

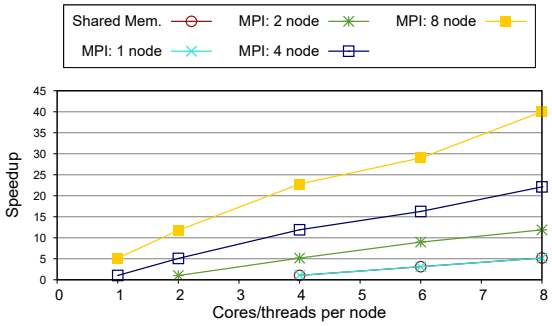
Figure 10: Speedup of the Mandelbrot *use case with equal and adjusted number of replicas.*

strongly benefits from very high throughput and very low latency networks, such as InfiniBand or Intel Omni-Path.

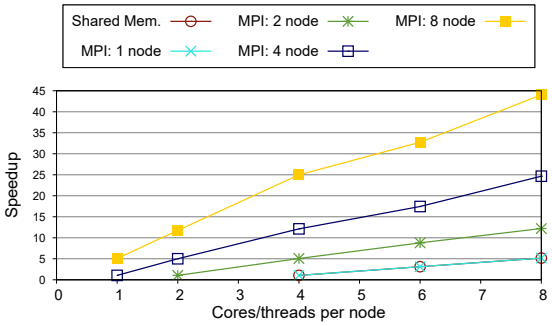
From these experiments, we can also conclude that the proposed GrPPI interface can aid in implementing distributed stream scientific applications at the expense of negligible overheads. In a separate experiment, we evaluated the overhead introduced by GrPPI w.r.t. using MPI directly. This overhead was less than 0.1 %.

VI.2 Productivity analysis

To analyze the productivity and usability of the GrPPI pattern interface and the new MPI back end, we make use of the Lizard analyzer tool [28] to obtain two well-known metrics: Lines of Code (LOCs) and the McCabe's Cyclomatic Complexity Number (CCN) [29]. Basically, we leverage these metrics to analyze the different versions of

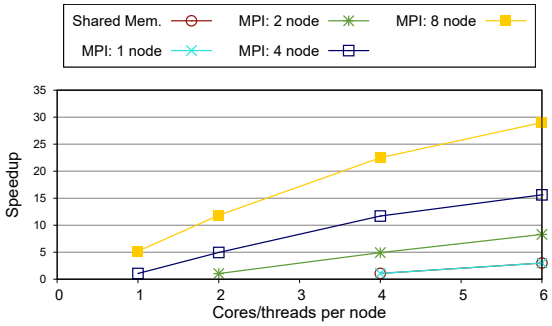


(a) Speedup using two-sided communications.

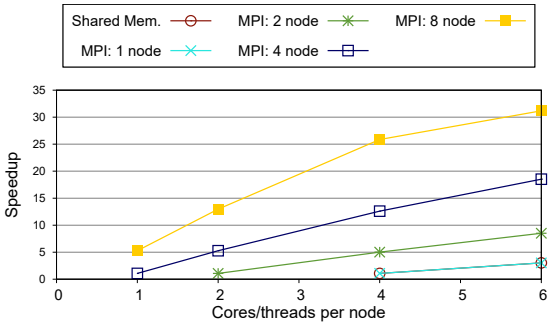


(b) Speedup using one-sided communications.

Figure 11: Speedup of the MANDELBROT *use case on* TUCAN.



(a) Speedup using two-sided communications.



(b) Speedup using one-sided communications.

Figure 12: Speedup of the MANDELBROT *use case on* TINTORRUM.

the MANDELBROT use case, i.e., with and without using the GRPPI interface. Table 1 summarizes the percentage of additional LOCs introduced into the sequential source code in order to implement the parallel versions using MPI and the GRPPI interface, along with their corresponding CCNs. As observed, implementing more complex compositions via MPI leads to larger and more complex source codes, while for GRPPI the number of additional LOCs remains constant. This difference is mainly due to the communication queues required to implement the Farm pattern. Focusing on GRPPI, we observe that the parallelization effort is almost negligible: even the most complex composition increases nearly 4.2 % the LOCs. Also, switching GRPPI to use a particular execution policy just needs changing a single argument in the pattern function call. Regarding the cyclomatic complexity for MPI, we observe that their CCNs are roughly proportional to the LOCs percentage increase. On the contrary, the GRPPI interface has constant CCNs for all Pipeline compositions.

Finally, we perform a side-by-side comparison between the Boost MPI using one-side primitives and the GRPPI interfaces for implementing a simplified version of the Mandelbrot use case. As can be seen in Listing 13(a), the MPI implementation clearly distinguishes the instructions and communications that have to be performed by each of the processes.[2] On the other hand, the GRPPI code, shown in Listing 13(b), focuses more on the application algorithm structure rather than on the inter-process communications. In a nutshell, although both interfaces provide high-level interfaces, we conclude that the pattern implementations offered by GRPPI help improving both productivity and maintainability. This is mainly to the algorithm encapsulation provided by the design pattern approach.

[2]For the sake of simplicity, we have replaced the use of queues by synchronous communications, so no computations overlap communications in this case.

Table 1: Percentage of additional LOCs w.r.t. the sequential version and CCNs for the Pipeline compositions.

Pipeline composition	% additional LOCs		CCN	
	MPI	GRPPI	MPI	GRPPI
(p|p|p|p)	+10.3 %	+4.2 %	9	7
(p|f|p|p)	+130.4 %	+4.2 %	38	7
(p|p|f|p)	+130.4 %	+4.2 %	38	7
(p|f|f|p)	+185.8 %	+4.2 %	58	7

VII. CONCLUSIONS

In this paper, we have extended GRPPI, a generic and reusable parallel pattern interface, with a new MPI back end, which enables the execution of the Pipeline and Farm stream patterns on distributed platforms. To support hybrid scenarios, the back end also combines an intra-node shared-memory execution policy that if needed, is

```cpp
std::vector<color> image;
if (world.rank() == 0) { // Zoom generator
  for(int frame= 0; frame < num_frames; frame++) {
    zoom-= zoom * 0.1;
    world.send(world.rank()+1, 0, zoom);
  }
}
else if (world.rank() == 1) { // Mandelbrot stage
  for(int frame= 0; frame < num_frames; frame++) {
    world.recv(world.rank()-1, 0, zoom);
    image = mandelbrot(height, width, poi_x, poi_y, zoom);
    world.send(world.rank()+1, 0, image);
  }
}
else if (world.rank() == 2) { // Blur stage
  for(int frame= 0; frame < num_frames; frame++) {
    world.recv(world.rank()-1, 0, image);
    image = blur(height, width, kernel, image);
    world.send(world.rank()+1, 0, image);
  }
}
else if (world.rank() == 3) { // Consuming stage
  for(int frame= 0; frame < num_frames; frame++) {
    world.recv(world.rank()-1, 0, image);
    save_bmp(height, width, image);
  }
}
```

(a) Boost MPI implementation.

```cpp
int frame= 0;
int num_frames= 1000;
// Main pipeline
grppi::pipeline(mpi_exec,
  // Zoom generation
  [&]() -> std::experimental::optional<double> {
    if (frame++ == num_frames) return {};
      zoom-= zoom * 0.1;
    return zoom;
  },
  // Farm mandelbrot stage
  grppi::farm(4,
    [&](auto zoom){
      return mandelbrot(height, width, poi_x, poi_y, zoom
          );
    }
  ),
  // Farm blur stage
  grppi::farm(4,
    [&](auto image){
      return blur(height, width, kernel, image);
    }
  ),
  // Consuming stage
  [&](auto image){
    save_bmp(height, width, image);
  }
);
```

(b) GrPPI implementation.

Figure 13: Boost MPI and GrPPI *implementations of the Mandelbrot use case.*

used to run multiple Pipeline and/or Farm operators inside the same MPI process. In general, the compact design of GrPPI facilitates the development of data streaming applications, improving flexibility and portability, while exploiting both inter- and intra-node parallelism. Additionally, we have provided support for two communication mechanisms among stream operators on top of two-sided and one-sided MPI primitives.

As demonstrated throughout the experimental evaluation, both the Video-App and the Mandelbrot use cases, implemented with the distributed Pipeline and Farm patterns, attains considerable speedup gains compared with the corresponding sequential version. We have also demonstrated that the proposed communication mechanisms attain the same performance using an Ethernet interconnection network; while for high-throughput and low latency networks that support RDMA, the one-sided mode outperforms the two-sided communication mode. In any case, as seen through the evaluation, it is always important to balance Pipeline stages according to the stage workloads in order to better exploit the available resources. We also proved that leveraging GrPPI reduces considerably the number of LOCs and cyclomatic complexity with respect to implementing them using directly MPI. Additionally, thanks to the qualitative comparison of the two high-level interfaces, GrPPI and MPI, we conclude that GrPPI leads to more structured and readable codes, and thus, improves their general maintainability. All in all, the implementation of a distributed back end by means of MPI has been mainly motivated by the scaling needs and development of new programming models for DaSP scientific applications. Furthermore, our interest with this back end comes from the wide adoption of MPI in today's supercomputers, which currently has no standard support for stream processing [16]. For these reasons, we believe that the presented back end can greatly aid in developing stream applications in C++.

As future work, we plan to support new algorithms for mapping operators onto processes and introduce a new operator to allow users to replace the by default execution policy of a concrete operator in a Pipeline stage. We also plan to implement other streaming patterns, such as the Filter or Stream-Reduce constructions, within the MPI execution policy and improve the communication queues to use one-sided MPI communications.

Acknoledgments

This work was partially supported by the EU project ICT 644235 "RePhrase: REfactoring Parallel Heterogeneous Resource-Aware Applications" and the project TIN2013-41350-P "Scalable Data Management Techniques for High-End Computing Systems" from the *Ministerio de Economía y Competitividad*, Spain.

References

References

[1] G. Fox, S. Jha, L. Ramakrishnan, Stream2016: Streaming requirements, experience, applications and middleware workshop, Tech. rep., Lawrence Berkeley National Laboratory (10 2016). doi:10.2172/1344785.

[2] S. Kamburugamuve, S. Ekanayake, M. Pathirage, G. Fox, Towards high performance processing of streaming data in large data centers, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, Chicago, USA, 2016, pp. 1637–1644. doi:10.1109/IPDPSW.2016.103.

[3] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, The MIT Press, Massachusetts, USA, 2014.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, Parallel Programming in OpenMP, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[5] F. Cappello, D. Etiemble, Mpi versus mpi+openmp on ibm sp for the nas benchmarks, in: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC '00, IEEE Computer Society, Washington, DC, USA, 2000, pp. 12–12.
URL http://dl.acm.org/citation.cfm?id=370049.370071

[6] M. McCool, J. Reinders, A. Robison, Structured Parallel Programming: Patterns for Efficient Computation, 1st Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[7] A. Marco, D. Marco, K. Peter, T. Massimo, Fastflow: HighLevel and Efficient Streaming on Multicore, Wiley-Blackwell, 2017, Ch. 13, pp. 261–280. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119332015.ch13, doi:10.1002/9781119332015.ch13.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119332015.ch13

[8] P. Ciechanowicz, M. Poldner, H. Kuchen, The Münster Skeleton Library Muesli: A comprehensive overview, ERCIS Working Papers 7, University of Münster, European Research Center for Information Systems (ERCIS) (2009).
URL https://ideas.repec.org/p/zbw/ercisw/7.html

[9] A. Ernstsson, L. Li, C. Kessler, Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems, International Journal of Parallel Programming 46 (1) (2018) 62–80. doi:10.1007/s10766-017-0490-5.
URL https://doi.org/10.1007/s10766-017-0490-5

[10] D. del Rio Astorga, M. F. Dolz, J. Fernández, J. D. García, A generic parallel pattern interface for stream and data processing, Concurrency and Computation: Practice and Experience Online (2017) e4175–n/a. doi:10.1002/cpe.4175.

[11] J. F. Muñoz, M. F. Dolz, D. del Rio Astorga, J. P. Cepeda, J. D. García, Supporting mpi-distributed stream parallel patterns in grppi, in: Proceedings of the 25th European MPI Users' Group Meeting, EuroMPI'18, ACM, New York, NY, USA, 2018, pp. 17:1–17:10. doi:10.1145/3236367.3236380.
URL http://doi.acm.org/10.1145/3236367.3236380

[12] S. T. Allen, M. Jankowski, P. Pathirana, Storm Applied: Strategies for Real-time Event Processing, 1st Edition, Manning Publications Co., Greenwich, CT, USA, 2015.

[13] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65. doi:10.1145/2934664.

[14] S. Papp, The Definitive Guide to Apache Flink: Next Generation Data Processing, 1st Edition, Apress, Berkely, CA, USA, 2016.

[15] W. Thies, M. Karczmarek, S. Amarasinghe, Streamit: A language for streaming applications, in: R. N. Horspool (Ed.), Compiler Construction, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 179–196.

[16] I. B. Peng, S. Markidis, R. Gioiosa, G. Kestor, E. Laure, MPI streams for HPC applications, CoRR abs/1708.01306. arXiv:1708.01306.
URL http://arxiv.org/abs/1708.01306

[17] I. B. Peng, S. Markidis, E. Laure, D. Holmes, M. Bull, A data streaming model in mpi, in: Proceedings of the 3rd Workshop on Exascale MPI, ExaMPI '15, ACM, New York, NY, USA, 2015, pp. 2:1–2:10. doi:10.1145/2831129.2831131.
URL http://doi.acm.org/10.1145/2831129.2831131

[18] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, M. Torquati, Targeting distributed systems in fastflow, in: Proceedings of the 18th International Conference on Parallel Processing Workshops, Euro-Par'12, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 47–56. doi:10.1007/978-3-642-36949-0_7.
URL http://dx.doi.org/10.1007/978-3-642-36949-0_7

[19] R. Loogen, Y. Ortega-mallén, R. Peña marí, Parallel functional programming in eden, J. Funct. Program. 15 (3) (2005) 431–475. doi:10.1017/S0956796805005526.
URL https://doi.org/10.1017/S0956796805005526

[20] J. F. Ferreira, J. L. Sobral, A. J. Proenca, Jaskel: a java skeleton-based framework for structured cluster and grid computing, in: Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on, Vol. 1, IEEE, Singapore, 2006, pp. 4 pp.–304. doi:10.1109/CCGRID.2006.65.

[21] K. Matsuzaki, H. Iwasaki, K. Emoto, Z. Hu, A library of constructive skeletons for sequential style of parallel programming, in: Proceedings of the 1st International Conference on Scalable Information Systems, InfoScale '06, ACM, New York, NY, USA, 2006. doi:10.1145/1146847.1146860.
URL http://doi.acm.org/10.1145/1146847.1146860

[22] P. Ciechanowicz, M. Poldner, H. Kuchen, The Münster Skeleton Library Muesli: A comprehensive overview, Working Papers, ERCIS - European Research Center for Information Systems 7, Westf. Wilhelms-Univ., Münster (2009).
URL http://hdl.handle.net/10419/58419

[23] E. Alba, G. Luque, J. Garcia-Nieto, G. Ordonez, G. Leguizamon, MALLBA: a Software Library to Design Efficient Optimisation Algorithms, Int. J. Innov. Comput. Appl. 1 (1) (2007) 74–85. doi:10.1504/IJICA.2007.013403.

[24] K. Fuerlinger, T. Fuchs, R. Kowalewski, DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms, in: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), IEEE, Sydney, Australia, 2016, pp. 983–990. doi:10.1109/HPCC-SmartCity-DSS.2016.0140.

[25] D. Gregor, M. Troyer, Boost MPI (2017).
URL https://www.boost.org/doc/libs/1_65_0/doc/html/mpi.html

[26] W. Gropp, R. Thakur, An evaluation of implementation options for mpi one-sided communication, in: B. Di Martino, D. Kranzlmüller, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 415–424.

[27] C. Computer Architecture, S. (ARCOS), Generic Reusable Parallel Pattern Interface - GrPPI, https://github.com/arcosuc3m/grppi/, online; accessed 5 May 2018 (2018).

[28] Terry Yin, Lizard: an Cyclomatic Complexity Analyzer Tool, https://github.com/terryyin/lizard, online; accessed 5 May 2018 (2018).

[29] T. J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. 2 (4) (1976) 308–320. doi:10.1109/TSE.1976.233837.