This is a postprint version of the following published document:

DOI: https://doi.org/10.1016/j.parco.2019.03.004

© 2019 Elsevier B.V. All rights reserved.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
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 GaPPI, 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 GaPPI 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 GaPPI 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 GaPPI overheads.

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

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 (GaPPI) [10] with a new MPI back end, which enables the execution of some streaming patterns on distributed platforms. Basically, GaPPI 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 GaPPI MPI execution policy for distributed and hybrid environments for both Pipeline and Farm parallel patterns.
- We describe the design of the GaPPI 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 GaPPI 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 GaPPI 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 GaPPI 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 analyzes 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 MPISstream, a prototype library implemented atop MPI, which provides an interface to existing MPI applications to adopt the streaming model [16][17]. Basically, MPISstream 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 GaPPI MPI back end is twofold. On the one hand, GaPPI 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 GaPPI MPI back end transparently enables the execution of streaming C++ application on HPC distributed platforms, as GaPPI 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 GaPPI 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 \( f_n(f_{n-1}(\ldots f_1(x_1)\ldots)) \). 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 “\( (P|F|F|P) \)”. 

![Diagram](image)

**Figure 1: Pipeline and Farm pattern diagrams.**

### III.2 GrPPI, a generic parallel pattern interface

Regarding the parallel pattern interface, we have leveraged GaPPI, 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 GaPPI 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 GaPPI library.

![GaPPI Architecture](image)

**Figure 2: GaPPI architecture.**

Listing 3 shows both Pipeline and Farm GaPPI 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.

```cpp
template <typename E, typename G> void farm(int num_replicas, G &generator, E &operator);
```

![Pipeline and Farm GaPPI pattern interfaces](image)

**Figure 3: Pipeline and Farm GaPPI 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
executing policy, we determine the need for communication situations 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 GaPPI patterns, we identify four different communication scenarios:

- Single-Producer, Single-Consumer (SPSC): two consecutive Pipeline stages (p1p).
- Single-Producer, Multiple-Consumer (SPMC): a Pipeline stage to a set of Farm replicas (p1f).
- Multiple-Producer, Single-Consumer (MPSC): a set of Farm replicas to a Pipeline stage (f1p).
- Multiple-Producer, Multiple-Consumer (MPMC): a set Farm replicas to another set of Farm replicas (f1f).

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_send/receive primitives. Therefore, no controller threads in the 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. As shown in the MPMC queue producer side, the queue creates a controller thread per producer process 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.

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,
IV MPI COMMUNICATION QUEUES

IV.2 One-sided communication queues

(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.

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 \((\text{rank}, \text{offset}, \text{length})\) 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 \(\text{length}\) bytes, starting at \(\text{offset}\) from the data window of the process with rank \(\text{rank}\).

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 \texttt{MPI_Allgather} primitive. Afterwards, all the processes involved in a queue call the \texttt{MPI_Comm_create_group} function providing a group generated using the received bitmaps. Finally, all the processes involved in this new communicator initialize their respective windows using the collective communication primitive \texttt{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
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 GaPPI 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 GaPPI 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, GaPPI 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).

As observed, the execution policy constructors receive the program arguments directly. Additionally, the second argument determines the preferred communication mode using an enum 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: GaPPI MPI execution policy class.

```cpp
template<typename Type> class ParallelExecutionNative {
public:
  ParallelExecution(int argc, char** argv, CommMode comm_mode = one_sided,
                   LocalPolicy local_exec_policy = ParallelExecutionNative());
  void parallel_execution(int argc, char** argv, CommMode comm_mode = one_sided,
                           LocalPolicy local_exec_policy = ParallelExecutionNative());
  void parallel_execution_native();
};
```

As shown in Listing 2, the GaPPI 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.

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:

\[
opp = \frac{\text{num}_{\text{ops}}}{\text{num}_{\text{procs}}}
\]  

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:

\[
\text{Range}(\text{rank}) = \begin{cases} 
  \{\text{rank} \times \text{opp}, ((\text{rank} + 1) \times \text{opp}) - 1\} & \text{if } \text{rank} \neq \text{num}_{\text{procs}} - 1 \\{\text{rank} \times \text{opp}, \text{num}_{\text{ops}} - 1\} & \text{if } \text{rank} = \text{num}_{\text{procs}} - 1 
\end{cases}
\]

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 GaPPI 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.
VI. EXPERIMENTAL EVALUATION

In this section, we perform an experimental evaluation of the GaPPI 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 GaPPI MPI back end built on top of GaPPI 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 GaPPI 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).
VI EXPERIMENTAL EVALUATION

VI.1 Performance analysis of Tucan platform using the InfiniBand network. As it can be noticed, the hardware level.

directly involved in the communications, as it RDMA is handled at 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.

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 GaPPI 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 GaPPI 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 GaPPI.

VI.1 Performance analysis of Video-App

In this section, we evaluate the performance of Video-App with using images of size 400 x 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 bi-section 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 GaPPI 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.
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 GaPPI 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 GaPPI 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 GaPPI 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
the **Mandelbrot** use case, i.e., with and without using the **GaPPI** 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 **GaPPI** interface, along with their corresponding CCNs. As observed, implementing more complex compositions via MPI leads to larger and more complex source codes, while for **GaPPI** the number of additional LOCs remains constant. This difference is mainly due to the communication queues required to implement the **Farm** pattern. Focusing on **GaPPI**, we observe that the parallelization effort is almost negligible: even the most complex composition increases nearly 4.2% the LOCs. Also, switching **GaPPI** to use a particular execution policy just needs changing a single argument in the pattern function call. Regarding the cyclo-matic complexity for MPI, we observe that their CCNs are roughly proportional to the LOCs percentage increase. On the contrary, the **GaPPI** 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 **GaPPI** 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. On the other hand, the **GaPPI** 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 **GaPPI** help improving both productivity and maintainability. This is mainly to the algorithm encapsulation provided by the design pattern approach.

**VII. Conclusions**

In this paper, we have extended **GaPPI**, 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

---

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

<table>
<thead>
<tr>
<th>Pipeline composition</th>
<th>% additional LOCs</th>
<th>CCN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>MPI</strong></td>
<td><strong>GaPPI</strong></td>
</tr>
<tr>
<td>(p</td>
<td>p</td>
<td>p</td>
</tr>
<tr>
<td>(p</td>
<td>f</td>
<td>p</td>
</tr>
<tr>
<td>(p</td>
<td>p</td>
<td>f</td>
</tr>
<tr>
<td>(p</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>

---

<sup>2</sup>For the sake of simplicity, we have replaced the use of queues by synchronous communications, so no computations overlap communications in this case.
used to run multiple Pipeline and/or Farm operators inside the same MPI process. In general, the compact design of GaPPI 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-AFP 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 GaPPI 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, GaPPI and MPI, we conclude that GaPPI 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.

Acknowledgments

This work was partially supported by the EU project ICT 644235 “RePhrase: REfactoring Parallel Heterogeneous Resource-Aware Management Techniques for High-End Computing Systems” from the Ministerio de Economía y Competitividad, Spain.

References

References


