Paving the Way Towards High-Level Parallel Pattern Interfaces for Data Stream Processing

David del Rio Astorga[1], Manuel F. Dolz[1], Javier Fernández[1] and J. Daniel García[1]

[1] *Universidad Carlos III de Madrid, 28911–Leganés, Spain*

The emergence of data stream applications has posed a number of new challenges to existing infrastructures, processing engines and programming models. In this sense, high-level interfaces, encapsulating algorithmic aspects in pattern-based constructions, have considerable reduced the development and parallelization efforts of these type of applications. An example of parallel pattern interface is GrPPI, a C++ generic high-level library that acts as a layer between developers and existing parallel programming frameworks, such as C++ threads, OpenMP and Intel TBB. In this paper, we complement the basic patterns supported by GrPPI with the new stream operators Split-Join and Window, and the advanced parallel patterns Stream-Pool, Windowed-Farm and Stream-Iterator for the aforementioned back ends. Thanks to these new stream operators, complex compositions among streaming patterns can be expressed. On the other hand, the collection of advanced patterns allows users to tackle some domain-specific applications, ranging from the evolutionary to the real-time computing areas, where compositions of basic patterns are not capable of fully mimicking the algorithmic behavior of their original sequential codes. The experimental evaluation of the new advanced patterns and the stream operators on a set of domain-specific use-cases, using different back ends and pattern-specific parameters, reports remarkable performance gains with respect to the sequential versions. Additionally, we demonstrate the benefits of the GrPPI pattern interface from the usability, flexibility and maintainability points of view.
[1]

1 Introduction

With the rise of big data, services and instruments, such as mobile devices, social media and sensor networks, are constantly producing huge amounts of data [22]. This extreme on-line generated data growth poses profound challenges to existing processing engines, programming models and infrastructures. In this sense, existing big data application models, such as MapReduce, have become popular for batch processing. Nevertheless, these models cannot fulfill the strict requirements of low latency and high throughput demanded by data streaming applications (DaSP). To address these issues, the DaSP paradigm is a more adequate approach to deal with their real-time requirements [8]. Basically, the DaSP model considers that the data is not fully available nor stored in disk or memory, but rather they continuously arrive from one or more streams. The idea behind this paradigm is that data has to be processed as soon as it is received.

[1]This is a post-print of an article published in Journal Future generation computer systems. The final authenticated version is available online at: https://doi.org/10.1016/j.future.2018.05.011

In general, current DaSP applications are mostly based on consecutively applying independent operators on an input data stream. In this sense, the throughput of these application can be accelerated by parallelizing these filter operations using traditional approaches, however, incorporating such optimizations is usually a time-consuming and error-prone task. An alternative to simplify this procedure is the use of pattern-based programming models that encapsulate algorithmic aspects using a building blocks approach [25]. Fundamentally, parallel patterns offer a way to implement robust, readable and portable solutions, while hide away the complexity behind concurrency mechanisms, e.g., thread management, synchronizations or data sharing. Numerous examples of pattern-based programming frameworks, such as SkePU [16], FastFlow [5] or Intel TBB [28], can be found in the literature. Nevertheless, most of these frameworks are not generic enough nor offer unified pattern interfaces [10]. To tackle these issues, the recent interface GrPPI [14], accommodates a unified layer of generic and reusable parallel patterns on the top of existing execution environments and pattern-based frameworks. Basically, GrPPI allows users to implement parallel applications without having a deep understanding of existing parallel programming frameworks or third-party interfaces and, thus, relieves the development and maintainability efforts. In contrast to other object-oriented implementations in the literature, this interface uses C++ template meta-programming techniques in order to provide generic interfaces of the patterns without incurring in significant runtime overheads.

However, we find that the core patterns currently offered by GrPPI do not match or have to be composed among them in a very complex way in order to comply with some domain-specific algorithms. Also, certain data stream applications cannot be easily modeled with the basic streaming patterns provided by this interface. For these reasons, we extend the basic patterns with *i)* stream operators, intended to modify the stream flow in pattern compositions, and *ii)* advanced patterns, modeling some domain-specific algorithms that cannot be represented using basic patterns or simple compositions. Specifically, this paper contributes with the following:

- We provide an initial set of stream operators in GrPPI, Window and Split-Join, able to modify the stream flow to conform windows and model flow graphs, respectively.
- We complement the core patterns supported by GrPPI with the advanced stream parallel patterns: Stream-Pool, Windowed-Farm, and Stream-Iterator.
- We demonstrate the flexibility and the composability of the stream operators and advanced patterns in the GrPPI context.
- We assess the patterns usability with respect to the number of lines of code (LOCs) and cyclomatic complexity required to implement a set of use cases. Additionally, we perform a side-by-side comparison of the two high-level interfaces: GrPPI and Intel TBB.
- We evaluate the performance gains by using these patterns on the set of domain-specific use cases and a real streaming application with varying parallelism degree configurations and problem-specific parameters.

In general, this paper extends the results presented in [15] with *i)* the inclusion of the stream operators, *ii)* the different policies for such stream operators, *iii)* the stream-oriented Stream-Pool pattern and *iv)* the evaluation of the usability and maintainability of the proposed constructions.

The remainder of this paper is organized as follows. Section 3 states the formal definition of the stream operators and advanced parallel patterns supported by GrPPI. Section 4 describes the interface adopted for the new operators and patterns presented in this paper. Section 5 evaluates these patterns from the usability and performance points of view under four different use cases. Section 2 revisits some related works about parallel programming frameworks and domain-specific patterns. Section 6 gives a few concluding remarks and future works.

2 Related Work

In the literature, numerous efforts of parallel patterns targeted to modern architectures for developing data stream applications have been proposed. These efforts can be classified in the two following categories: *i)* data stream processing engines and *ii)* pattern-based parallel programming frameworks.

In the recent years, several engines for shared-memory and distributed DaSP have been developed, such as Storm [6], Spark [31], Flink [26] and StreamIt [30]. Basically, applications implemented using these engines are represented as direct flow graphs, where nodes represent operators and the edges the stream flow. According to how the operators, or nodes in the graph, and the edges are defined, different and complex operations for filtering, splitting and joining streams can be performed. However, they do not fully support the operators nor advanced parallel patterns proposed in this paper. For instance, the Window operator is not supported in Storm nor StreamIt, but provide mechanisms that allow developers to define the processing logic and windowing policy. However, Spark and Flink do provide support for this operator natively. Indeed, Flink offers the *count-based* and the *time-based* policies, while Spark only supports the latter. Focusing on the Split-Join operator, we realize it is present in all these frameworks given that it is an inherent operation for building large complex data stream graphs. In general, we detect that, while operators are partially supported by these frameworks, the advanced patterns have to be manually implemented.

On the other hand, we review some current parallel-pattern interfaces from the state-of-the-art oriented to: *i)* multi-core processors, e.g., Intel Thread Building Blocks (TBB) [28], RaftLib [9] or Kanga [21]; *ii)* heterogeneous architectures, such as, SkePU [16], which allows hybrid CPU–GPU configurations; and *iii)* distributed platforms, e.g., the Münster Skeleton Library (Muesli) [17] and CnC [12]. Simultaneously, standardized interfaces are being progressively developed. This is the case of C++ STL algorithms, available in the forthcoming C++17, that start defining parallel versions of already existing STL algorithms [19]. Similar implementations to the parallel STL can also be found as third-party libraries, e.g., HPX [20] and GrPPI [14]. All in all, we observe that streaming patterns are not fully adopted by these frameworks, either because they only target data-intensive patterns or because they not allow to build complex flow graphs. For instance, Intel TBB offers a wide range of function nodes that permit the large graph constructions, but the logic behind the Window and Split-Join operators should be provided by the user. Other interfaces, such as SkePU or Muesli, offer a very limited range of stream patterns, thus do not give the opportunity to build advanced data streaming applications.

With regard to advanced parallel patterns, we see that none of the aforementioned DaSP engines nor high-level pattern interfaces natively supply advanced patterns and, if needed, these constructions should be implemented from scratch. In this sense, we only find that some pattern-based frameworks in the literature have pushed forward the development of such high-level patterns. For instance, the FastFlow [5] library recently provided the Stream-Pool [4] and the Windowed-Farm [13] patterns, two common structures in evolutionary and stream-intensive applications, respectively. On the other hand, the MALLBA library [2] accommodates a collection of high-level skeletons for combinatorial optimization which deals with parallelism in a user-friendly and efficient manner. Specifically, MALLBA provides implementations of Particle Swarm, Ant Colony Optimizations and Scatter Search for gene selection problems and metaheuristics [3].

In any case, the reviewed approaches do not yet offer generic enough interfaces to be easily leveraged when developing DaSP applications. Throughout this paper and following this motivation, we have provided generic high-level interfaces for this type of applications by means of extending the collection of GrPPI stream patterns with some stream operators and advanced parallel patterns.

3 Data stream parallel patterns

Patterns have been generally defined as recurring strategies for solving problems from a wide spectrum of areas, such as architecture, object-oriented programming and software architecture [23, 25]. In our case, we take advantage of parallel software design patterns, since they provide a mechanism to encapsulate algorithmic features and are able to make applications more robust, portable and maintainable. Also, if these patterns are properly tuned, they can achieve a good balance between parallel scalability and data locality.

In this sense, several solutions from the state-of-the-art offer collections of basic parallel patterns as a “building blocks” modeling strategy for developing stream processing applications. However, while many of the algorithms found in DaSP applications match directly those patterns, there exist situations in which those have to be composed among them in order to comply with the algorithm requirements. On the other hand, modeling complex data flow graphs is infeasible only with the basic patterns, given they are not able to modify the stream, either by splitting or joining it. To tackle these issues, we extend the basic patterns with *i)* stream operators, designed to work cooperatively with regular patterns, and *ii)* advanced patterns, modeling some domain-specific algorithms that cannot be represented using basic patterns or simple compositions. It is worth noting that the stream operators and the advanced patterns have been inspired from the literature in stream processing [7] and the RePhrase project [1], respectively.

3.1 Stream operators

In this section we formally describe the Split-Join and the Window stream operators along with their different configurations.

Split-Join This stream operator allows the programmer to express applications as compositions of basic patterns in directed flow graphs, where the nodes and edges represent kernels and data streams, respectively. Specifically, this operator distributes, in a first Split phase, a data stream into different substreams, which can be processed in parallel applying distinct transformations. Afterwards, a Join phase combines again the substreams into a single one (see Figure 1a). This operator is characterized by the different distribution and combination policies that can be used in both Split and Join phases. The supported policies are the following:

Duplication This policy duplicates the data items appearing on the input stream to each of the different substreams. In other words, when an item arrives at the input stream, it is copied into every substream. By definition, this policy can only be applied in the Split phase of the Split-Join operator.

Round-robin This policy can be applied in both Split and Join phases. If applied on the Split phase, the data items appearing on the input stream are delivered following a round-robin policy onto the substreams. On the other hand, if used on the Join phase, the items delivered at the end of the substreams are combined together into the main data stream employing the same policy. In this distribution policy, the slice size, i.e. number of consecutive items that should be taken from the source stream, can be freely configured.

Window This stream logic operator consumes input items and delivers tuples or "windows" to the output stream and allows a kernel to work with a concrete segment of input data (see Figure 1b). Note that this operator is, by nature, *stateful*, as the results produced are affected by the internal data structures used for its processing logic. Depending on how the windows are internally managed by the operator, different windowing policies can be set. The following policies can be defined based on how many and which items are part of a same window.

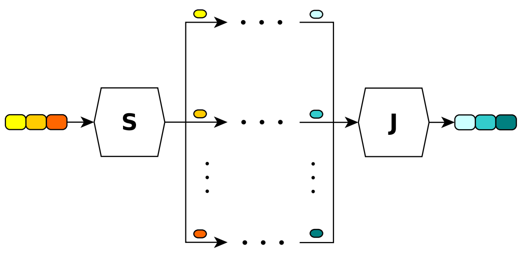
Count-based This policy is characterized by managing windows of fixed size, i.e., capable of holding up a maximum number of items. Note that the user should specify the window size. The rationale of this policy is the following: when a new item arrives at the input stream, it is included in the window and the oldest is flushed. As soon as the window is complete, it is delivered to the output stream.

Delta-based This policy requires a δ threshold value and a monotonic increasing (Δ) attribute included in the input items. With these parameters, the *delta-based* policy is able to build a window of variable size. The items conforming a window are only those whose difference between the Δ attribute of the previous items and the current one is less or equal than the given δ threshold. Similar to the *count-based* policy, when a window is built, it is forwarded to the output stream.

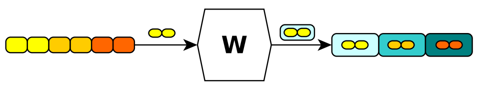
Time-based This policy keeps a internal wall-clock and labels each input item with a timestamp indicating their arrival time. Also, it requires the users to specify a time threshold (τ). This information allows the policy to build windows including the items that arrived in the last time, as specified by τ. For instance, a threshold of 60 s would conform windows including items that arrived in the last minute.

Punctuation-based This policy works with a *punctuation* value that indicates the end of a window. In other words, it delivers a new window each time a new item matching the *punctuation* value is received. For instance, considering a stream of words belonging to a text and using the "." character as for the punctuation value, this policy would conform windows containing the sentences in the text.

Note that the *count-*, *delta-* and *time-based* windowing policies support an sliding factor, i.e., the number of items in the window w_i that are also part of the window w_{i+1}.



(a) Split-Join.



(b) Window.

Figure 1: Stream operators.

It is important to remark that the proposed stream operators have been designed to work cooperatively with other patterns. The idea of this operators is to modify the stream flow, either by agglomerating stream items into windows or splitting/joining the main stream for performing different computations in the substreams.

3.2 Advanced stream parallel patterns

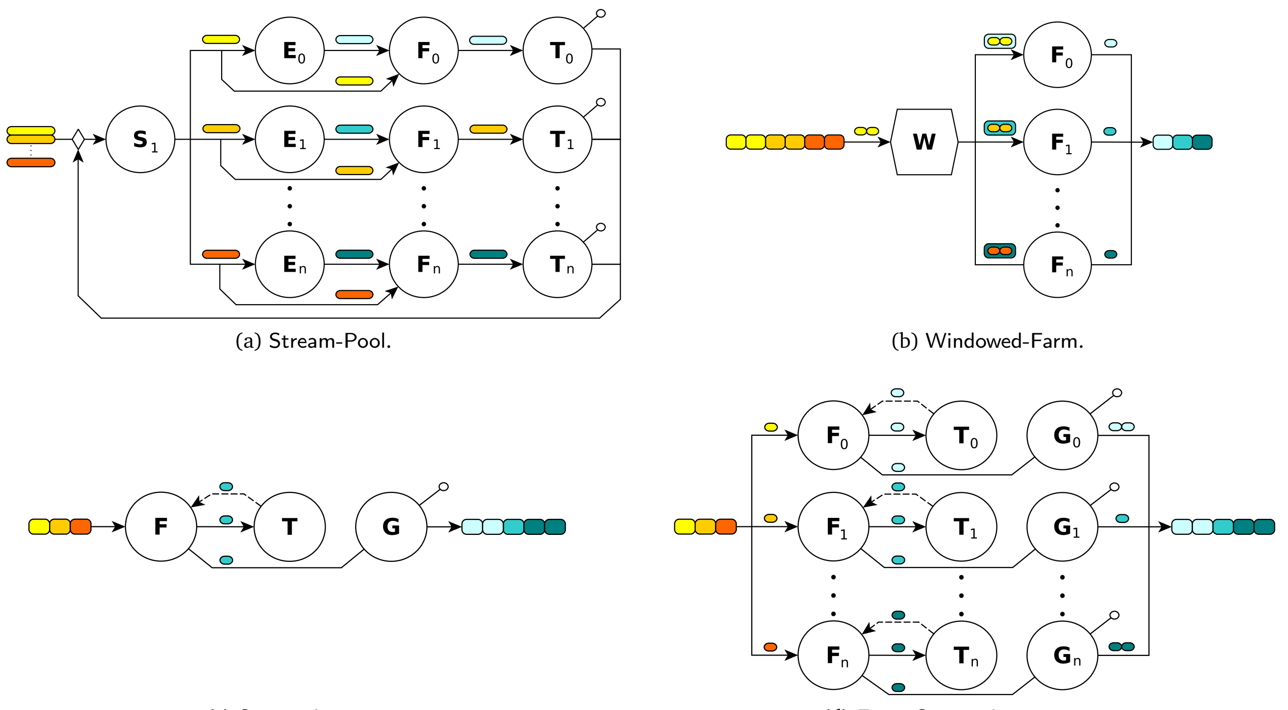
In this section, we describe some advanced stream parallel patterns, designed for those scenarios in which the basic patterns do not match any of these constructions or have to be composed in a very complex way. This occurs in many domain-specific algorithms coming from the evolutionary and symbolic computing [18] domain, wireless sensor networks algorithms [11] or real-time processing engines [27]. Therefore, we determine the need for supporting advanced patterns in order to simplify the development of sophisticated algorithms related to the aforementioned application domains.

In the following, we describe formally three new parallel patterns that can be eventually incorporated during the parallelization task of such applications: Stream-Pool, Windowed-Farm and Stream-Iterator.

Stream-Pool This pattern models the evolution of a population of individuals matching many evolutionary computing algorithms in the state-of-the-art [4]. Specifically, the Stream-Pool pattern is comprised of four different functions that are applied iteratively to the individuals of type α belonging to a population P managed as a stream (see Figure 2a). First, the *selection* function S: $\alpha^* \to \alpha^*$ selects a subset of individuals belonging to P. Next, the selected individuals are processed by means of the *evolution* function E: $\alpha^* \to \alpha^*$, which may produce any number of new or modified individuals. The resulting set of individuals, computed by E, are filtered through a *filter* function F: $\alpha^* \to \alpha^*$, and eventually inserted into the input stream (population). Finally, the *termination* function T: $\alpha^* \to \{true, false\}$ determines in each iteration whether the evolution process should be finished or continued. To guarantee the correctness of the parallel version of this pattern, the functions E, F and T should be pure, i.e., they can be computed in parallel with no side effects.

Windowed-Farm This pattern delivers "windows" of processed items to the output stream. Basically, this pattern applies the function F over consecutive contiguous collections of x input items of type α and delivers the resulting windows of y items of type β to the output stream (see Figure 2b). Note that this pattern simplifies the composition of the Window and Farm patterns. Also, the windows produced by this pattern benefit from the same windowing policies provided by the Window stream operator. The parallelization of this pattern requires a pure function F: $\alpha^* \to \beta^*$ for processing windows.

Stream-Iterator This pattern is intended to recurrently compute the pure function F: $\alpha \to \alpha$ on a single stream input item until a specific condition, determined by the boolean function T: $\alpha \to \{true, false\}$, is met. Additionally, in each iteration the result of the function F is delivered to the output stream, depending on a boolean output guard function G: $\alpha \to \{true, false\}$ (see Figure 2c). Note that this pattern, due to its nature, does not provide any parallelism degree by itself and can be classified as a pattern modifier. Therefore, the parallel version of this construction is only achieved when it is composed with some other core stream pattern, e.g., using Farm or Pipeline as for the function F. An example of Stream-Iterator composed with a Farm pattern is shown in Figure 2d.



(a) Stream-Pool. (b) Windowed-Farm.
(c) Stream-Iterator. (d) Farm–Stream-Iterator.

Figure 2: Advanced parallel patterns.

4 GrPPI operators and advanced stream pattern interfaces

In this section, we extend our generic and reusable parallel pattern interface (GrPPI) for C++ applications, previously presented in [14], with the stream operators and advanced parallel patterns respectively described in Sections 3.1 and 3.2. In general, GrPPI takes full advantage of modern C++ features, metaprogramming concepts and generic programming to act as switch between the parallel programming models OpenMP, C++ threads and Intel TBB. Its design allows users to leverage the aforementioned execution frameworks just in a single and compact interface, hiding away the complexity behind the use of concurrency mechanisms with negligible overheads. Furthermore, the modularity of GrPPI permits to easily integrate new patterns, while compose them to arrange more complex ones. Thanks to these properties, GrPPI can be used to implement a wide range of existing stream processing and data-intensive applications with relative small efforts, having as a result portable codes that can be executed on multiple platforms.

In the subsequent sections we introduce the GrPPI interfaces for the proposed stream operators and advanced patterns. Furthermore, we also demonstrate its composability through a series of examples.

4.1 Stream operators interfaces

This section describes in detail the proposed interfaces for the Split-Join and Window stream operators.

Split-Join The GrPPI interface designed for the Split-Join operator, shown in Listing 1, receives the split policy (`split_policy`) and a list of transforming functions (`transformers`) that should be applied to each of the substreams. In this sense, the number of substreams of the Split-Join operator is determined by the number of functions received as arguments. Note that a transformation function can be a single function or a given pattern composition.

Specifically, during the Split phase, the items are taken from the input stream and distributed among the substreams according to the policy set by the user. Afterwards, different processing entities compute the transformations related to their substreams. In the final Join phase, the resulting items of the substreams are combined together into a single one following a round-robin policy.[2]

Listing 1: Split-Join interface.

```cpp
template <typename SP, typename ... T>
auto split_join(SP &split_policy, T ... transformers);
```

In the current implementation we provide support for both *duplicate* and *round-robin* policies (see Listing 2). Nevertheless, the set of policies can be eventually extended by means of declaring a new class implementing the functions: `set_num_streams` and `get_next_streams`. The first function is intended to set the total number of substreams, while the second is used get the IDs of the substreams to where the source stream items should be forwarded. The substreams IDs are determined by the order in which the transforming functions are placed in the Split-Join operator call.

Listing 2: Splitting policies interfaces for the Split-Join operator.

```cpp
// Duplication split policy
auto duplicate();
// Round-robin split policy
auto round_robin(int number);
```

Window The interface proposed for the Window operator, described in Listing 3, receives the windowing policy as a single argument. Basically, this stream operator takes the items from the input stream and stores them into an internal buffer. Next, depending on the selected policy, the buffer is delivered to the output stream as soon as a window is formed.

Listing 3: Window interface.

```cpp
template <typename WP>
auto window(WP &window_policy);
```

[2]Note that while the current Split-Join operator only supports round-robin as for the Join policy, in the future we plan to extend this interface to support other policies.

In the current implementation, we support four different windowing policies: *count-based*, *delta-based*, *time-based* and *punctuation-based* (see Listing 4). The behavior of these policies can tuned as desired by setting a collection of window-specific parameters, e.g., window size, punctuation value, sliding, etc. Following the same approach used for the splitting policies in the Split-Join operator, we also give the user the opportunity to implement customized policies. To make this possible, the internal implementation of the Window operator internally uses the functions: `add_item` and `get_window`, which entirely dictate the behavior of the operator. The first receives the input item and returns a boolean determining whether the window is formed or not. The second function returns the window once it is formed. Thus, providing the operator with different implementations of these functions allows the incorporation of new policies.

Listing 4: Windowing policies interfaces for the Window operator.

```cpp
// Count-based windowing policy
template <typename ItemType>
auto count_window(int w_size, int slide);
// Delta-based windowing policy
template <typename ItemType>
auto delta_window(int delta_value, int slide);
// Time-based windowing policy
template <typename ItemType>
auto time_window(int time_size, int slide);
// Punctuation-based windowing policy
template <typename ItemType>
auto punctuation_window(ItemType punctuation_value);
```

4.2 Advanced stream pattern interfaces

Next, we describe in detail the GrPPI interfaces for the proposed advanced stream parallel patterns: Stream-Pool, Windowed-Farm and Stream-Iterator.

Stream-Pool The GrPPI interface designed for the Stream-Pool pattern, shown in Listing 5, receives the execution model, the population (`popul`), the selection (`select`), evolving (`evolve`), filtering (`filter`) and termination (`term`) functions. Initially, the selection takes individuals from the original population and introduces them into the input stream of the pattern. Afterwards, the different processing entities take the individuals from the input stream and apply the evolve, termination and filter functions. If the termination condition is met, the pattern finalizes its execution and returns the population with the resulting individuals. Otherwise, depending on the filter function, an evolved or an original individual is introduced again into the input stream.

The parallelism of this pattern is controlled via the execution model parameter, which can be set to operate in sequential or in parallel, through the different supported frameworks; e.g. to use C++ threads, the parameter should be set to `parallel_execution_thr`. In this case, any execution model can optionally receive, as an argument, the number of entities to be used for the parallel execution, e.g., `parallel_execution_thr{6}` would use 6 worker threads. If this argument is not given, the interface uses by default as many threads as the total number of cores in the platform.

Listing 5: Stream-Pool interface.

```cpp
template <typename EM, typename P, typename S, typename E, typename F, typename T>
void stream_pool(EM exec_mod, P &popul, S &&select, E &&evolve, F &&filt, T &&term);
```

Windowed-Farm The interface for the Windowed-Farm pattern, described in Listing 6, receives the execution model, the stream consumer (`in`), the Farm (`transformer`) and the producer (`out`) functions. This pattern also receives the windowing policy. In this sense, this pattern is a simplified interface for a GrPPI composition of a Pipeline containing a Window operator and a Farm pattern. Specifically, the `in` function reads from the input stream as many items as required to fill the window buffer. Next, this buffer is forwarded to one of the concurrent entities, which will compute the function `task` in a Farm-like fashion. Therefore, the parallel implementation of this pattern is offered by the Farm construction. Finally, the items collections resulting from the `task` function are delivered to the output stream. Note that, depending on the user requirements, this pattern can deliver processed windows in an ordered way by properly configuring the execution model.

Listing 6: Windowed-Farm interface.

```cpp
template <typename EM, typename I, typename F, typename O, typename WP>
void windowed_farm(EM exec_mod, I &&in, F &&transformer, O &&out, WP &window_policy);
```

Stream-Iterator The GrPPI interface for the Stream-Iterator pattern, detailed in Listing 7, takes the execution model, the stream consumer (`in`), the kernel (`transformer`) and the producer (`out`) functions. This pattern also receives two boolean functions: the termination (`term`) and output guard (`guard`) functions. In the first step, the `in` function reads items from the input stream and a worker thread executes the kernel `transformer` function for each item. Next, the termination function `term` is evaluated with the resulting item to determine if the kernel should be re-executed on the same input item. Additionally, the output `guard` function decides whether an item should be delivered to the output stream or not.

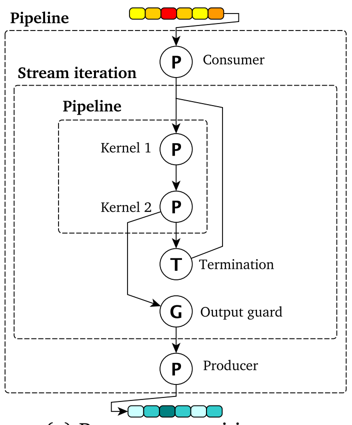
Listing 7: Stream-Iterator interface.

```
template <typename EM, typename I, typename F, typename O, typename T, typename G>
void stream_iteration(EM exec_mod, I &&in, F &&transformer, O &&out, T &&term, G &&guard);
```

Note that interfaces presented for these patterns are intended to work as a standalone functions, however, the GrPPI interface also offers composable interfaces that allow them to be used as part of other patterns, e.g. Pipeline or Farm. In those cases, the parameters related to the execution model, consumer and producer functions are automatically inherited from the outer pattern.

4.3 Pattern composability

As previously stated, the stream operators with basic and advanced patterns can be composed among them to match complex algorithms and procedures that may appear in DaSP applications. In this section, we explore the pattern composability features offered by the proposed constructions, stream operators and advanced patterns, using two simple application examples.



(a) Pattern compositions.

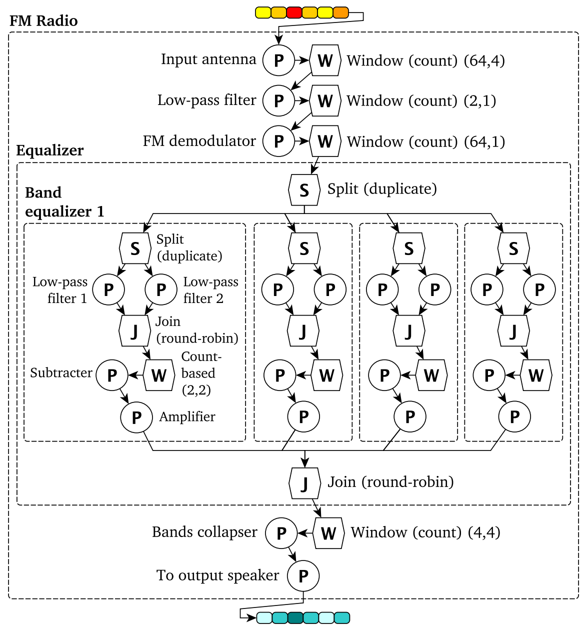
```
using namespace grppi;
...
pipeline(parallel_execution_thr{}
  [&]() -> optional<int> { // Consumer function
    auto value = read_value(is);
    return (value > 0) ? value : {};
  },
  stream_iteration(
    pipeline( // Kernel function
      [](int e){ return 3*e; },
      [](int e){ return e-1; }
    ),
    [](int e){ return e<100; }, // Termination function
    [](int e){ return e%2==0;} // Output guard function
  ),
  [&](int e){ os << e << endl; } // Producer function
);
```

(b) Implementation.

Figure 3: Example of Pipeline-Stream-Iterator-Pipeline composition in GrPPI.

The first application example reads the integers stored in a file, processes them until a threshold is reached and outputs the results according to a guard condition (see Figure 3a). To express this application in GrPPI terms, we make use of Pipeline-Stream-Iterator-Pipeline composition. As can be seen in Listing 3b, the consumer and producer functions are part of the outer Pipeline ends, while the computation itself is performed by means of the Stream-Iterator pattern. As stated in the previous section, the parallelism of the Stream-Iterator pattern is only obtained when it is composed with a parallel construction, e.g., Farm or Pipeline. In this case, we have leveraged a Pipeline as for the kernel computation, where two different threads simultaneously execute both stages. Recall that the termination function controls the number of times that an item is processed through the inner Pipeline, while the guard function determines which results should be forwarded to the producer function. It is also worth to remark that the `optional` return type in the consumer lambda function is used to indicate the end of the stream when constructed without arguments.

To extend the GrPPI pattern composability demonstration, we leverage the FM-Radio as a real study case of streaming application inspired by an example part of the StreamIt programming model [30] (see Figure 4a). Concretely, we implement the FM-Radio software taking advantage of the stream operators and basic GrPPI patterns. Basically, this application receives as for the input stream the signal from an external antenna and produces a new processed signal that is connected to a speaker. As can be seen in Listing 4b, the program is structured as a main Pipeline whose first two stages are: a band-pass filter to tune in the desired frequency and a



(a) Pattern compositions.

```cpp
using namespace grppi;
...
// FM Radio
pipeline(e,
  [&]() -> optional<float> {...}, // Input antenna
  window(count_window<float>(64,4)),
  [&coeff](auto window) {...}, // Low-pass filter
  window(count_window<float>(2,1)),
  [&mGain](auto window) {...}, // FM demodulator
  window(count_window<float>(64,1)),
  split_join(duplicate{}, // Equalizer (4 bands)
    pipeline( // Band equalizer 1
      split_join(duplicate{}, // Band-pass filter
        [&coeffs](auto win) {...},// Low-pass filter 1
        [&coeffs](auto win) {...} // Low-pass filter 2
      ),
      window(count_window<float>(2,2)),
      [](auto w) {...}, // Subtracter
      [&eqGain](float window) {...} // Amplifier
    ),
    ... // remaining band equalizers
  ),
  window(count_window<float>(4,4)),
  [](auto window) {...}, // Bands collapser
  [&](float f) {...} // To output speaker
);
```

(b) Implementation.

Figure 4: Pattern composition and implementation of the FM-Radio in GrPPI.

demodulator. Observe that these sequential stages are followed by Window operators in charge of conforming count-based windows. Next, the main Pipeline is continued with an equalizer expressed with a Split-Join operator, where each branch fine-tunes the gain of a specific frequency range. The last Pipeline stages collapse all bands and emit the processed signal to an external speaker.

As can be seen, thanks to the presented stream operators and advanced patterns we are able to implement complex data-flow graphs with relatively small efforts. Simultaneously, the GrPPI interface enhances expressiveness and simplicity due to the hierarchical block-level abstraction offered by the parallel patterns. All in all, our goal for designing these constructs in a composable way is to improve both programmability and readability of complex streaming applications.

5 Evaluation

In this section, we perform an experimental evaluation of the stream operators and advanced GrPPI patterns from the usability and performance points of view. To do so, we use the following hardware and software components:

- **Target platform**. The evaluation has been carried out on a server platform comprised of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM. The OS is a Linux Ubuntu 14.04.2 LTS with the kernel 3.13.0-57.
- **Software**. To implement the proposed pattern interfaces, we leveraged the execution environments C++11 threads and OpenMP 4.5, and the pattern-based parallel framework Intel Threading Building Blocks (TBB) as for the GrPPI back ends. The C++ compiler used to assemble GrPPI was GCC v6.2 which already supports the C++14 standard.
- **Use cases**. To evaluate both the stream operators and advanced patterns we use four different use cases targeting problems from different domains, implemented using directly the aforementioned execution environments and the GrPPI interface.

  FM-Radio This study case is a signal processing application emulating the software of a FM radio and it is employed to evaluate the basic patterns and the stream operators. To better analyze the behavior of these patterns, we have implemented five different versions of the FM-Radio application with varying number of equalizer bands, from 1 to 5.

  TSP This benchmark solves the *traveling salesman problem* (TSP) using a regular evolutionary algorithm and it is intended to analyze the behavior of the Stream-Pool pattern. Specifically, this NP-problem computes the shortest possible route among different cities, visiting them only once and returning to the origin city.

Sensor This streaming application computes average window values of the readings from an emulated sensor and it is used to evaluate the performance of the Windowed-Farm pattern.

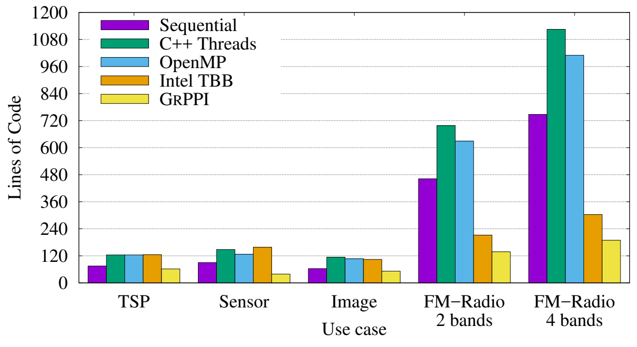
Image This stream-oriented benchmark reduces the resolution of images appearing in the input stream and delivers images with concrete resolutions to the output stream. Basically this benchmark is employed to evaluate the performance of the Stream-Iterator pattern.

In the following sections, we analyze the usability and the performance of the GrPPI stream operators and advanced patterns using the above-mentioned benchmarks with varying configurations of parallelism degree, problem size and execution frameworks.

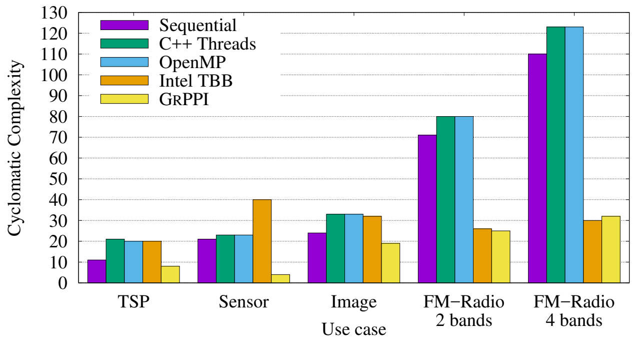
5.1 Usability analysis

To analyze the usability and flexibility of the proposed pattern interface, we make use of the Lizard analyzer tool [29] to obtain two well-known metrics: Lines of Code (LOCs) and the McCabe's Cyclomatic Complexity Number (CCN) [24]. Basically, we leverage these metrics to analyze the different use case versions, i.e., with and without using our GrPPI interface. Figure 5a shows the LOCs required to implement the parallel versions of the use case algorithms directly using the execution frameworks and the GrPPI interface. As observed, the TSP, Sensor and Image are simple use cases whose sequential versions require about 100 LOCs, while the FM-Radio is a more complex application with roughly 500 LOCs. Focusing on the simple use cases, we detect that the codes directly parallelized with the supported frameworks require almost twice the LOCs of the sequential versions, as none of them intrinsically implement the proposed high-level advanced patterns. On the contrary, the LOCs using GrPPI are significantly reduced with respect to the sequential code. Looking at the FM-Radio benchmark, we find out that for C++ threads and OpenMP, the LOCs increase by almost 50 % compared with the sequential version. This is given by the fact that the synchronization and management control mechanisms are not natively provided by these frameworks and have to be implemented by the user. Also, using a higher number of equalizer bands entails an increse of the LOCs. Differently, the high-level Intel TBB and GrPPI abstractions inherently incorporate these mechanisms and lead to smaller and more maintainable codes.

Looking at the cyclomatic complexity, detailed in Figure 5b, the CCNs related to TSP and Image use cases are roughly proportional to their LOCs. Nevertheless, the Image case does not follow this behavior as the Intel TBB framework requires the user to explicitly implement the windowing management mechanisms, entailing an increase in its complexity. Focusing on the FM-Radio with both 2 and 4 equalizer bands, we also detect that LOCs and CCNs keep approximately the same proportions. Another observation is that both Intel TBB and GrPPI frameworks present the same CCNs, while other frameworks outperform their CCNs by a factor of fourfold.



(a) Lines of code.



(b) Cyclomatic complexity.

Figure 5: Lines of code and cyclomatic complexity of the use cases w.r.t different programming models.

Finally, we perform a side-by-side comparison between the GrPPI and Intel TBB interfaces implementing the FM-Radio use case. As can be seen in Listing 6a, the GrPPI code follows a comprehensible and readable structure, which clearly shows the compositions among basic patterns and stream operators. On the contrary, the Intel TBB code, shown in Listing 6b, is not as structured and easy to read as the GrPPI implementation. Thus, in order to properly understand the application behavior, users need to carefully analyze the data flow graph implicitly declared in the Intel TBB code. In a nutshell, although both interfaces provide high-level parallel interfaces, we conclude that GrPPI leads to more structured and readable codes, and thus, improves both usability and maintainability.

```cpp
using namespace grppi;
...
// FM Radio
pipeline(e,
  [&]() -> optional<float> {...}, // Input antenna
  window(count_window<float>(64,4)),
  [&coeff](auto window) {...}, // Low-pass filter
  window(count_window<float>(2,1)),
  [&mGain](auto window) {...}, // FM demodulator
  window(count_window<float>(64,1)),
  split_join(duplicate{}, // Equalizer (4 bands)
    pipeline( // Band equalizer 1
      split_join(duplicate{}, // Band-pass filter
        [&coeffs](auto win) {...},// Low-pass filter 1
        [&coeffs](auto win) {...} // Low-pass filter 2
      ),
      window(count_window<float>(2,2)),
      [](auto w) {...}, // Subtracter
      [&eqGain](float window) {...} // Amplifier
    ),
    ... // remaining band equalizers
  ),
  window(count_window<float>(4,4)),
  [](auto window) {...}, // Bands collapser
  [&](float f) {...} // To output speaker
);
```

(a) GRPPI implementation.

```cpp
// FM Radio
struct split1 {void operator()(const buf_t &i,
  multi_node::output_ports_type &op) {...}};
struct window1{void operator()(const float &v,
  window_node::output_ports_type &op) {...}};
...
// Input antenna node
source_node<float> antenna(g,[&](float &v) -> bool {...} );
window_node wind1(g,serial,window1()); // Window node
queue_node<buf_t> win_q(g); // Window queue node
multi_node spl1(g,unlimited,split1()); // Split node
queue_node<buf_t> queue_band(g); // Split queue node
...
make_edge(antenna,win1); // Window edge
make_edge(output_port<0>(win1),win_q1); // Window-Queue edge
make_edge(win_q1,lowp1); // Low-pass filter edge
...
make_edge(output_port<0>(win3), win_q3);
make_edge(win_q3,spl1); // Band equalizer splitter edge
make_edge(output_port<0>(spl1), band_q);
...
make_edge(sub1,amp1); // Amplifier edge
make_edge(amp1,input_port<0>(j1)); // Joiner edge
...
make_edge(j1,adder); // Band collapser edge
make_edge(adder,speaker); // Output speaker edge
```

(b) Intel TBB implementation.

Figure 6: GRPPI and Intel TBB implementations of the FM-Radio.

5.2 Performance analysis of the FM-Radio

In this section, we evaluate the presented stream GRPPI operators using the aforementioned FM-Radio use case. Figure 7 depicts the speedup of the versions of application with varying number of equalizer bands and using the available back ends. In this experiment it is important to consider that the performance attained by the FM-Radio versions cannot be compared among them, as the applications are intrinsically different and produce distinct results. With this in mind, we observe that both C++ threads and OpenMP approximately attained the same speedups. However, the performance obtained by Intel TBB is much lower and, given that the windowing management techniques are not natively supported by the framework, it is required to use additional graph nodes to accomplish the same business logic. A final inspection on the results using Intel TBB reveals that the versions using a higher number of equalizer bands attain better performance, as the proportion between the number of Window operators and the basic patterns is lower. Therefore, overheads related to the Window operators are counteracted with effective computations performed by the parallel patterns.

From this experiment, we can conclude that the stream operators composed with regular patterns greatly aid in developing complex constructions, always at the expense of evaluating the best execution environment. The best choice for the back end basically depends on the application nature and the target platform.

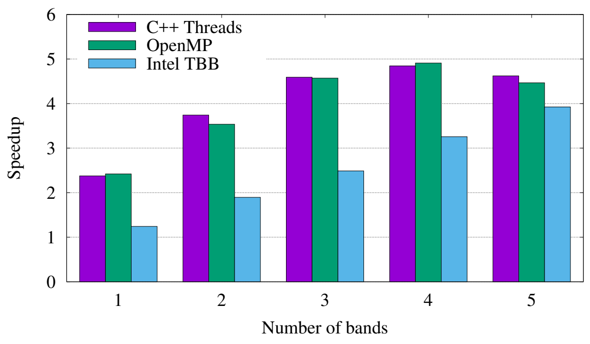
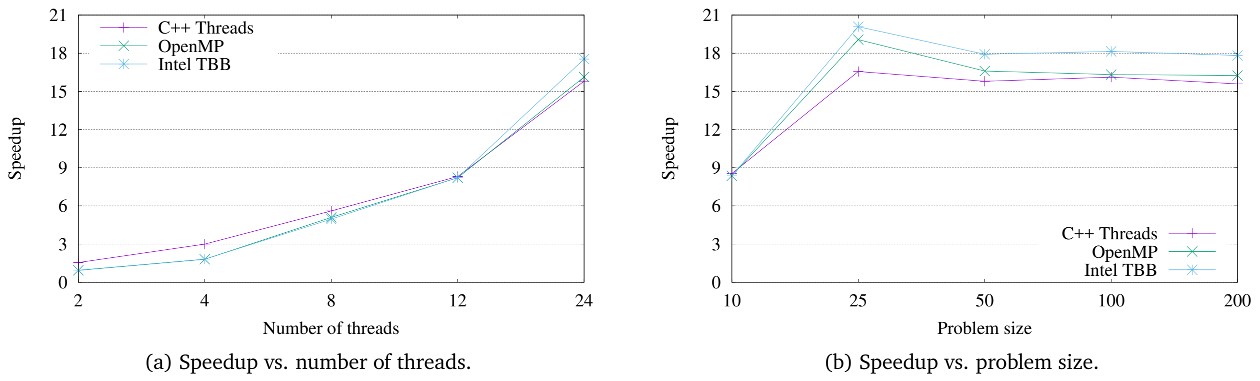


Figure 7: Speedup of the FM-Radio with varying number of bands.

5.3 Performance analysis of the Stream-Pool pattern

Next, we evaluate the Stream-Pool pattern on a benchmark that solves the TSP problem using an initial population of 50 individuals representing feasible routes. We also set the benchmark to perform a total of 200 iterations, each of them making 200 selections. Figure 8a shows the performance gains when varying the number of threads, from 2 to 24, and using the three available GrPPI back ends, C++ threads, OpenMP and Intel TBB, with respect to the sequential version. As can be seen, the speedup roughly increases at a linear rate when using a higher number of the number of threads for all frameworks. Concretely, we observe that between 2 and 12 threads, the efficiency is sustained in the range of 75 %–80 %. On the other hand, Intel TBB with 24 threads delivers an efficiency of 80 %, while the same in C++ threads and OpenMP lead to a slight decreased efficiency of 70 %. This is mainly due to the better resource usage made by the Intel TBB runtime scheduler.

As a complementary evaluation, we set the number of threads to 24 and vary the number of cities from 10 and 200. According to the results shown in Figure 8b, the parallelization overheads using 24 threads does not pay off for a small workloads, i.e., setting only 10 cities as for the problem size. However, when the problem size grows, the threads perform more effective computations and lead to a better application scalability. Also, we observe that above 50 cities the application becomes memory-bound due to the impact on handling the data structures representing feasible routes. Finally, it is also important to remark that those runtime-based frameworks (OpenMP and Intel TBB) provide better efficiency compared with the C++ threads execution environment.



(a) Speedup vs. number of threads. (b) Speedup vs. problem size.

Figure 8: Stream-Pool speedup varying with varying number of threads and problem size.

5.4 Performance analysis of the Windowed-Farm pattern

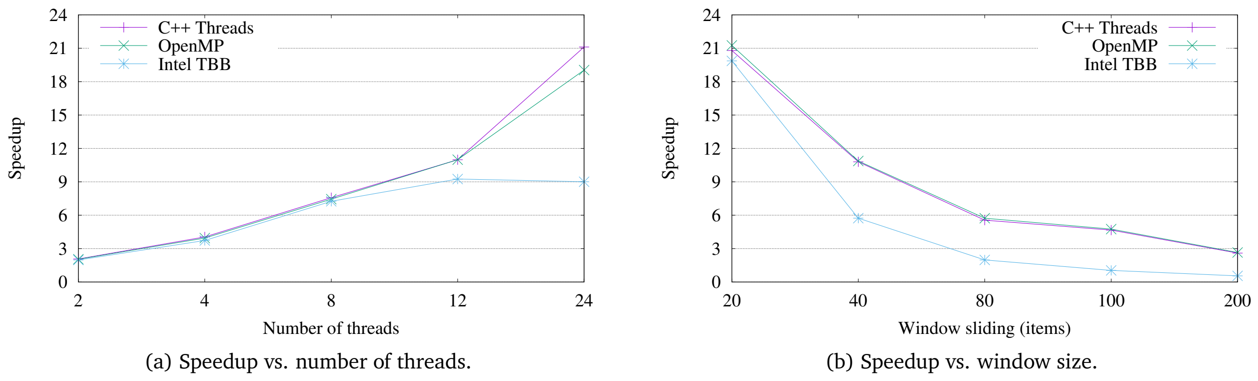
In this section, we evaluate the performance of the Windowed-Farm pattern using a synthetic benchmark that computes average window values from a sensor reading samples at 1 kHz. To carry out this evaluation, the following four subsections present different scenarios of the Windowed-Farm pattern using the proposed windowing management policies for the Window operator: *count-based*, *delta-based*, *time-based* and *punctuation-based*.

5.4.1 Analysis of the *count-based* windowing policy

As for the Windowed-Farm using the *count-based* policy, we set the window size to 100 elements with an overlap factor among windows of 90 %. Figure 9a shows the speedup when the number of threads increases from 2 to 24. The main observation is that both C++ threads and OpenMP frameworks scale with the increasing number of threads and behave similarly, given that the OpenMP runtime scheduler do not provide any major advantage over the C++ threads implementation in this concrete use case. This is because the internal Farm pattern leads, by nature, to well balanced workloads among threads. Note that a Farm is comprised of a pool of threads that constantly poll for items from the input stream and apply the same operation on them. On the other hand, we also observe an almost linear scaling for increasing number of threads. This is mainly caused because the Farm pattern can theoretically scale up to $\frac{T_f}{T_a}$, being T_f the computation time of the window average value and T_a the interarrival time of windows in the input stream. To demonstrate this strong scaling, we experimentally measured the computation time of the average function, which was, on average, 220 ms and the interarrival window time that was 10 ms. Therefore, applying the aforementioned formula, we get 22 as for the maximum theoretical speedup. In contrast, focusing on the Intel TBB back end, we observe that the application stops scaling from 12 threads on. The reason for this behavior is the same as discussed in Section 5.2, i.e., high-level pattern interfaces provided by Intel TBB do not intrinsically support any kind of windowing management policies. Therefore, performance overheads caused by the implementation of these policies in such a interface are non-negligible compared with those induced by using low-level frameworks, e.g., C++ threads and OpenMP. Concretely, the time required by the Window operator to generate a window is defined by $(T_i + T_o) \times window\_size$, where T_i and

T_o are the item interarrival time and the window management overheads per item, respectively. With this respect, the windowing overheads for Intel TBB are higher than for other frameworks and cause a general throughput decrease due to the use of such Window operator.

As an additional experiment using *count-based* policy, we evaluate the behavior of the same benchmark with varying window sizes and using 24 threads. As can be observed in Figure 9b, the speedup decreases for increasing window sizes, as the number of non-overlapping items among windows also increases. This basically occurs because the window interarrival time T_a increases, restricting proportionally the maximum parallelism degree.



(a) Speedup vs. number of threads. (b) Speedup vs. window size.

Figure 9: Windowed-Farm speedup with varying number of threads and window size using *count-based* windows.

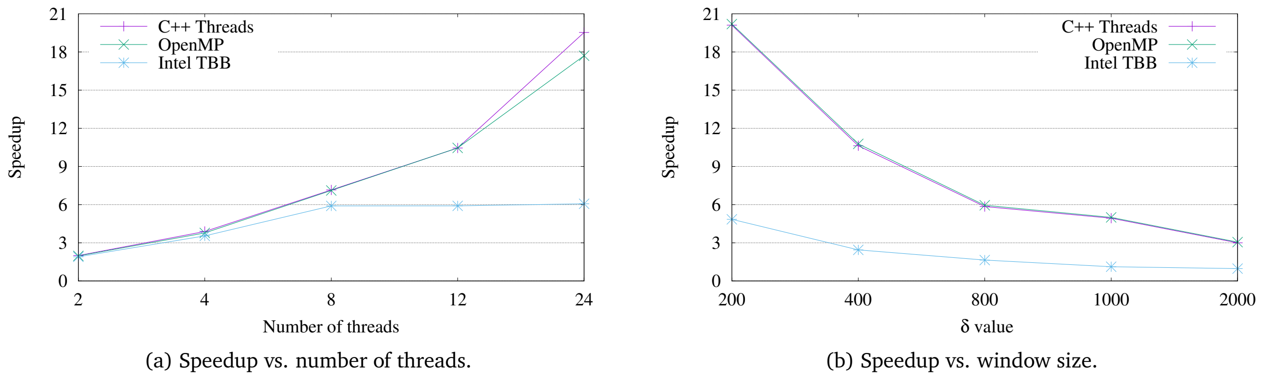
5.4.2 Analysis of the *delta-based* windowing policy

Regarding the evaluation of the Windowed-Farm pattern leveraging the *delta-based* windowing policy, we set the δ threshold of 1000 and a δ sliding of 150. Considering that the sensor of this synthetic application does not produce, by nature, monotonically increasing values, we have slightly modified the input stream to inject tuples containing the sensor reading and the Δ attribute to the Windowed-Farm pattern. Specifically, the Δ attribute equals to the sum of the previous samples. Figure 10a depicts the speedup of the different GrPPI back ends for different configurations of number of threads. As can be seen, the behavior of the *delta-based* policy is similar to that observed for *count-based*. In this sense, the speedups delivered by the respective C++ threads and OpenMP back ends go hand in hand and reach values of 18 and 20, respectively. This is mainly due to the similarities of both *count-based* and *delta-based* policies, where the windowing management overheads are mostly the same. Again, the lack of the Window operator in Intel TBB, produces notorious performance degradations from 8 threads on.

To extend this analysis, we perform additional experiment fixing the number of threads to 24 and varying the δ threshold value from 200 to 2000. Furthermore, we equal the δ sliding to the δ threshold value. Again, both C++ threads and OpenMP deliver similar speedups, with decrease for higher δ threshold values. This behavior is produced by the fact that a higher δ value tends to generate a lower number of windows with more items in each. Having in mind that processing entities compute at window-level, the concurrency degree at some point might not be fully exploited if the number of windows available to be processed is lower than the number of threads. The Intel TBB back end, however, suffers from considerable performance degradations, as the windows have to be internally handled by our Windowed-Farm pattern.

5.4.3 Analysis of the *time-based* windowing policy

For the evaluation of the Windowed-Farm using the *time-based* policy, we set the window time to 0.1 s with a sliding of 0.01 s. In this concrete case, we measure the number of windows that the application generates with respect to the expected ones, instead of the regular speedup metric. This is because the performance of the *time-based* windowing policy is directly related to the window time and sliding parameters, which for a same input stream determine the theoretical number of resulting windows. Any overhead related to the kernel function or to the window management, would make the application to deliver a higher number of windows (with less items per window) and to a general worse behavior. Considering our use case with a sensor producing 1000 samples/s, a window time of 0.1 s and a window sliding of 0.01 s, we would expect 100 windows/s with 100 elements per window. Figure 11a shows the percentage of number of windows with respect to the expected for varying number of threads. As can be observed, the sequential back end produces the worst results, as the percentages drawn are

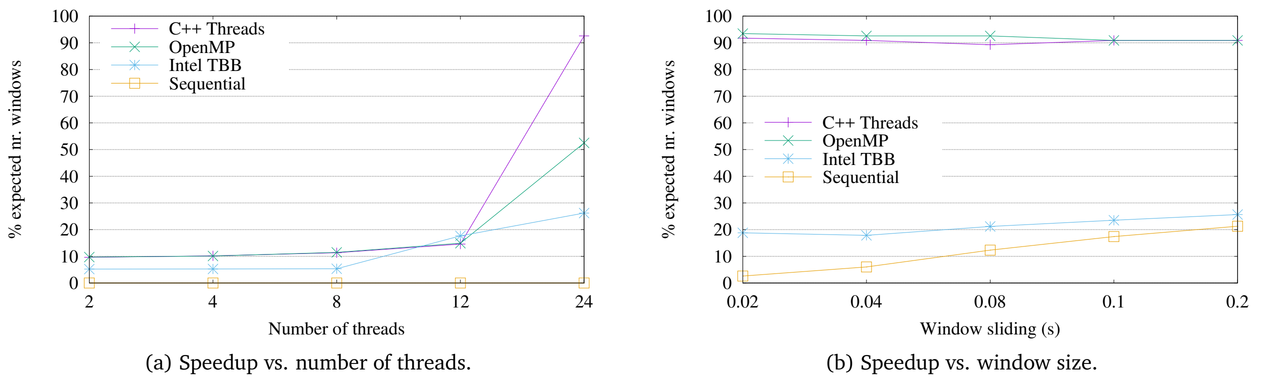


(a) Speedup vs. number of threads. (b) Speedup vs. window size.

Figure 10: Windowed-Farm speedup with varying number of threads and window size using *delta-based* windows.

close to 1%. Regarding the C++ threads, OpenMP and Intel TBB we detect as well that the behavior between 2 and 12 threads is far from the expected and ranges from 5% to 17%. This is due to the inherent congestions caused by the internal communication queues, as the worker threads are not able to process windows at the same pace that they arrive. Finally, focusing on the results for these back ends using 24 threads, we observe an in improved behavior for the C++ threads, OpenMP and Intel TBB, reaching a percentage of about 92%, 52% and 28%, respectively.

To gain more insights into the behavior of the *time-based* policy, we perform an additional experiment in which we fix the number of threads to 24 and vary the window sliding parameter from 0.02 to 0.2 seconds. As can be seen, when the sliding time increases, the behavior of the sequential version improves, as less windows per second have to be delivered. However, the Intel TBB version does not improve much and stays between 17% and 26%. In contrast, the C++ threads and OpenMP back ends deliver much better accuracy figures, around 92%, when the window sliding time increases. The reason for that is the overheads related to window management are almost negligible and affect, to a lesser extent, the production of windows in due time.



(a) Speedup vs. number of threads. (b) Speedup vs. window size.

Figure 11: Windowed-Farm speedup with varying number of threads and window size using *time-based* windows.

5.4.4 Analysis of the *punctuation-based* windowing policy

Finally, we evaluate the Windowed-Farm using the *punctuation-based* policy setting the delimiter value to 0. Specifically, we modified the sensor to randomly generate the value 0 with a probability of 0.01. Figure 12 shows the speedup attained by the benchmark when varying the number of threads from 2 to 24. Similar to the *count-based* and *delta-based* analyses, the speedups of both C++ threads and OpenMP using the *punctuation-based* windowing policy proportionally scales with the number of threads. This occurs because the delimiter value is generated enough for producing sufficient windows and feeding the worker entities. Conversely, Intel TBB has to deal with the same drawbacks detected during the evaluation of the previous policies, thus limiting the global speedup scaling beyond 8 threads.

In a nutshell, the different windowing policies presented for the Windowed-Farm, allows users to model a wide range of scenarios that can appear in DaSP applications. However, the user needs to be aware of the advantages

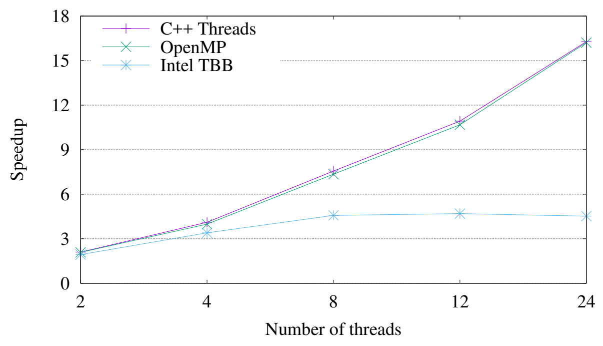
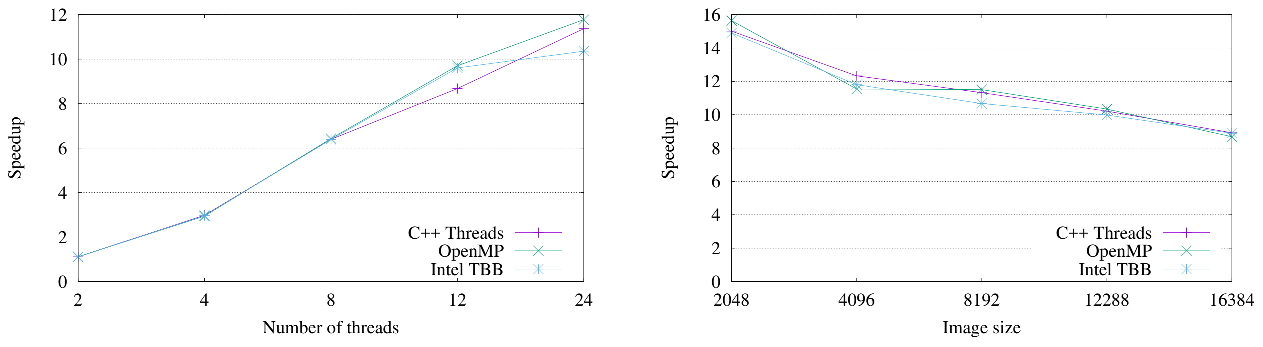


Figure 12: Windowed-Farm speedup with varying number of threads using *punctuation-based* windows

and drawbacks inherent to each policy and GrPPI back end.

5.5 Performance analysis of the Stream-Iterator pattern

Finally, we analyze the performance of the GrPPI Stream-Iterator pattern using the above-mentioned benchmark, in charge of processing square images and halving their sizes on each iteration until reaching concrete resolutions. Specifically, the size of the input images is fixed to 8,192 pixels, and the output images, for each input, have sizes of 128, 512 and 1,024. Figure 13a illustrates the benchmark speedup when varying the number of threads from 2 to 24 for the different GrPPI back ends. In this case, when the number of threads ranges between 2 and 12, the efficiency attained is roughly 75 %, while for 24 this is degraded to 48 % for all programming frameworks. This effect is mainly caused by the fact that each of the threads involved in the Farm pattern, used in the Stream-Iterator, are simultaneously accessing to different input images. Therefore, these memory accesses become a bottleneck due to constant cache misses when the threads perform the computation of the `task` function of the pattern. In general, these results suggest a memory bandwidth limitation in this particular benchmark.



(a) Speedup vs. number of threads. (b) Speedup vs. image size.

Figure 13: Stream-Iterator speedup with varying number of threads and image size.

To gain insights into the performance degradation detected in the previous analysis, we perform an additional experiment in which we set the number of threads to 24 and vary the input image sizes from 2,048 to 16,384. Figure 13b depicts the performance gains for the different execution frameworks when varying the image size in the preceding range. Again, we observe a slight speedup decrease for increasing image sizes, which confirms our prior impressions. As an example, if we assume 22 worker threads in the internal Farm pattern, individually processing images with resolution of $2{,}048 \times 2{,}048$ pixels (represented with matrices of integers), these require about 352 MiB of memory. Therefore, not fitting in any of the available cache levels and leading to an increased L2/L3 cache miss rate when they are simultaneously accessed. All in all, this issue is mainly due to the inherent memory-bound nature of this specific use case.

6 Conclusions

In this paper, we have complemented GrPPI, a generic and reusable parallel pattern interface, with the stream operators Window and Split-Join, and the advanced parallel patterns Stream-Pool, Windowed-Farm and Stream-Iterator, able to build complex DaSP applications. In this sense, the stream operators are able to modify the stream flow to conform windows and allowing to model graph structures. Conversely, the advanced patterns are intended to model some domain-specific algorithms that cannot be represented using complex compositions of basic patterns. Having these patterns implemented under the GrPPI umbrella, we are able to execute them in parallel using any of the currently supported back ends: C++ threads, OpenMP and Intel TBB. Furthermore, their compact design facilitates the development of complex streaming workflows while considerably reduces maintainability and portability efforts.

As demonstrated throughout the experimental evaluation, the use cases implemented with the proposed patterns and operators attain remarkable speedup gains compared with their corresponding sequential versions. Although, in some cases, the parallelism degree is limited by the application nature. We also proved that leveraging GrPPI reduces considerably the number of LOCs and cyclomatic complexity with respect to implementing them using directly the supported back ends. Additionally, thanks to the qualitative comparison among the two high-level interfaces, GrPPI and Intel TBB, we conclude that GrPPI leads to more structured and readable codes, and thus, improves their general maintainability. In summary, we believe that GrPPI stream processing interfaces, including stream operators, basic and advanced patterns, can greatly aid developers to implement complex applications offering a smoother learning curve than those required by other pattern-oriented frameworks.

As future work, we plan to support other advanced parallel patterns, such as the *keyed stream farm* and the *image convolution*. Also, we contemplate the *feedback* flow modifier and new policies for the Split-Join operator. Furthermore, we intend to include other execution environments as for the offered parallel frameworks, e.g., FastFlow or SkePU. An ultimate goal is to provide support for accelerators via CUDA Thrust and OpenCL SYCL. [3]

References

[1] RePhrase EU Project, D2.5: Advanced patterns. https://rephrase-eu.weebly.com/uploads/3/1/0/9/31098995/d2-5.pdf.

[2] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. Mallba: A library of skeletons for combinatorial optimisation. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, pages 927–932, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[3] Enrique Alba, Gabriel Luque, Jose Garcia-Nieto, Guillermo Ordonez, and Guillermo Leguizamon. MALLBA: a Software Library to Design Efficient Optimisation Algorithms. *Int. J. Innov. Comput. Appl.*, 1(1):74–85, April 2007.

[4] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Pool evolution: A parallel pattern for evolutionary and symbolic computing. *International Journal of Parallel Programming*, 44(3):531–551, 2016.

[5] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley & Sons, Inc., 2017.

[6] Sean T. Allen, Matthew Jankowski, and Peter Pathirana. *Storm Applied: Strategies for Real-time Event Processing*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[7] Henrique C. M. Andrade, Bugra Gedik, and Deepak S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, New York, NY, USA, 1st edition, 2014.

[8] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[3]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.

[9] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 96–105, New York, NY, USA, 2015. ACM.

[10] Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Department of Computer Science, Heriot-Watt University, December 2013.

[11] Doina Bucur, Giovanni Iacca, Giovanni Squillero, and Alberto Tonda. An evolutionary framework for routing protocol analysis in wireless sensor networks. In Anna I. Esparcia-Alcázar, editor, *Applications of Evolutionary Computation*, pages 1–11, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, August 2010.

[13] Tiziano De Matteis and Gabriele Mencagli. Parallel patterns for window-based stateful operators on data streams: An algorithmic skeleton approach. *International Journal of Parallel Programming*, 45(2):382–401, 2017.

[14] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, (Online):e4175–n/a, April 2017.

[15] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. Supporting Advanced Patterns in GrPPI, a Generic Parallel Pattern Interface. In Dora B. Heras and Luc Bougé, editors, *Euro-Par 2017: Parallel Processing Workshops*, pages 55–67, Cham, 2018. Springer International Publishing.

[16] Johan Enmyren and Christoph W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[17] Steffen Ernsting and Herbert Kuchen. Data parallel algorithmic skeletons with accelerator support. *International Journal of Parallel Programming*, 45(2):283–299, Apr 2017.

[18] Ewa Gajda-Zagórska. Multiobjective evolutionary strategy for finding neighbourhoods of pareto-optimal solutions. In Anna I. Esparcia-Alcázar, editor, *Applications of Evolutionary Computation*, pages 112–121, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[19] ISO/IEC. *Programming Languages – Technical Specification for C++ Extensions for Parallelism*, July 2015. ISO/IEC TS 19570:2015.

[20] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[21] D. Kist, B. Pinto, R. Bazo, A. R. D. Bois, and G. G. H. Cavalheiro. Kanga: A skeleton-based generic interface for parallel programming. In *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, pages 68–72, Oct 2015.

[22] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A.H. Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinskey&Company, 2011.

[23] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[24] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[25] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[26] Stefan Papp. *The Definitive Guide to Apache Flink: Next Generation Data Processing*. Apress, Berkely, CA, USA, 1st edition, 2016.

[27] Vladan Popovic, Kerem Seyid, Eliéva Pignat, Ömer Çogal, and Yusuf Leblebici. Multi-camera platform for panoramic real-time hdr video construction and rendering. *Journal of Real-Time Image Processing*, 12(4):697–708, 2016.

[28] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[29] Terry Yin. Lizard: an Cyclomatic Complexity Analyzer Tool. `https://github.com/terryyin/lizard`. Online; accessed 19 October 2017.

[30] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R. Nigel Horspool, editor, *Compiler Construction*, pages 179–196, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[31] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.