IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 24, 2019, accepted September 9, 2019, date of publication September 12, 2019,
date of current version September 25, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2941183*

# Raising the Parallel Abstraction Level for Streaming Analytics Applications

**GABRIELE MENCAGLI[1], MASSIMO TORQUATI[1], DALVAN GRIEBLER[2], MARCO DANELUTTO[1], AND LUIZ GUSTAVO L. FERNANDES[2]**

[1]Department of Computer Science, University of Pisa, I-56127 Pisa, Italy
[2]School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre 90619-900, Brazil

Corresponding author: Gabriele Mencagli (mencagli@di.unipi.it)

**ABSTRACT** In the stream processing domain, applications are represented by graphs of operators arbitrarily connected and filled with their business logic code. The APIs of existing Stream Processing Systems (SPSs) ease the development of transformations that recur in the streaming practice (e.g., filtering, aggregation and joins). In contrast, their *parallelism abstractions* are quite limited since they provide support to stateless operators only, or when the state is organized in a set of key-value pairs. This paper presents how the *parallel patterns* methodology can be revisited for sliding-window streaming analytics. Our vision fosters a design process of the application as *composition* and *nesting* of ready-to-use patterns provided through a C++17 *fluent* interface. Our prototype implements the run-time system of the patterns in the FastFlow parallel library expressing *thread-based parallelism*. The experimental analysis shows interesting outcomes. First, our pattern-based approach allows easy prototyping of different versions of the application, and the programmer can leverage nesting of patterns to increase performance (up to 37% in one of the two considered test-bed cases). Second, our FastFlow implementation outperforms (three times faster) the handmade porting of our patterns in popular JVM-based SPSs. Finally, in the concluding part of this paper, we explore the use of a *task-based* run-time system, by deriving interesting insights into how to make our patterns library suitable for multi backends.

**INDEX TERMS** Data stream processing, streaming analytics, sliding-window queries, parallel patterns, multicore programming.

## I. INTRODUCTION

The data deluge generated by our ever-more-connected world raises the need of easy-to-use frameworks able to efficiently process data streams in real-time. Such frameworks should provide high-level user-friendly programming interfaces for easing the developing of efficient streaming applications. Furthermore, they should enable the efficient execution on modern hardware, not only limited to clusters as in traditional systems like Apache Storm [1] and Apache Flink [2], but also on modern powerful scale-up servers equipped with tens of cores and terabytes of memory.

Despite this second direction has gained attention in the last years [3], [4], the first problem of programming abstractions lacks of recent research advancements. This is particularly true for what regards *parallelism abstractions*, which should leave the programmer free to express parallelism in a powerful and flexible way, by exploiting the interplay existing between the space of parallelism opportunities often available in stream processing programs (e.g., pipelining, task and data parallelism).

Parallelism in existing tools is easy to express for stateless operators, through replication directly enabled in the API by the user, or for *keyed streams*, that is when the stream is partitionable in logical substreams based on a key attribute. In our prior work [5], we proposed a set of *parallel patterns* targeting continuous analytics based on *sliding windows*. Such kind of queries are supported in the existing frameworks and represent an essential part of many streaming benchmarks [6]. In the paper in [5], the patterns were introduced and preliminarily evaluated through low-level experimentations for testing their individual properties.

The present paper extends our prior work in three novel directions. First, we study how to design a C++17 library called WindFlow[1] that exposes those patterns to the final users through a high-level interface and facilitates their

The associate editor coordinating the review of this manuscript and approving it for publication was Tomás F. Pena.

[1]GitHub link: https://github.com/ParaGroup/WindFlow

nesting to improve performance. Second, we show an implementation of this library in FastFlow [7], a C++ run-time environment for *thread-based parallelism* on multicores, and we present several pattern transformations to deal with some of the issues deriving from the choice of this parallelism model. Finally, in the concluding part of the paper, we discuss the impact of a different run-time system (based on *task-based parallelism*) and we identify some important issues that need to be addressed to obtain satisfactory performance.

More precisely, our main contributions are as follows:

- we show that our patterns can be developed in a compositional way by using simpler patterns to build more complex ones. This helps code modularity and reuse, and allows complex nested structures to be built in a bottom-up fashion to leverage different parallelism paradigms to improve performance (i.e. between sub-streams, between and within windows);
- we introduce a set of transformations of the parallel structure deriving from the patterns nesting, in order to simplify it by removing centralization points or by collapsing under-utilized entities into a single thread. Such transformations reveal useful for mitigating some of the issues raised by the adoption of the thread-based parallelism model of the FastFlow implementation, which is similar to the design of the most popular streaming frameworks (notably, Apache Flink and Apache Storm);
- the patterns have been emulated in Apache Flink and some of them in Apache Storm by using a proper composition of the available operators. The performance comparison in two test-based applications shows a significant throughput improvement and latency reduction by our C++ implementation;
- for one of the two proposed applications, we provide a preliminary study of the impact of using the Intel TBB library as an alternative run-time system, showing that this parallelism approach, which has known advantages in terms of dynamicity and load balancing, requires a proper tuning of the task granularity to demonstrate its full potential in terms of sustained throughput even if at the cost of increased latency.

This paper has the following organization. Next section presents the background. Section III introduces the building blocks of our FastFlow implementation. Section IV describes the API and the implementation of our patterns, with a set of transformations shown in Section V. Experiments are described in Section VI. Finally, Section VII reviews similar works and Section VIII draws the conclusions.

## II. BACKGROUND AND MOTIVATION

In this part, we describe the most characterizing features of the Data Stream Processing paradigm (DSP) by focusing on the programming abstractions provided in the existing Stream Processing Systems (SPSs) to express parallelism. Then, we recall the methodology of *Parallel Design Patterns* [8] traditionally applied in the Parallel Computing domain, and we discuss why a merge with the DSP paradigm is necessary

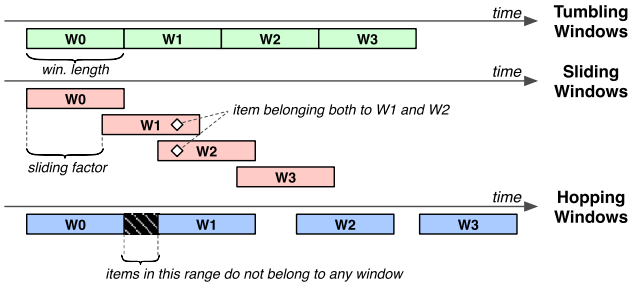



**FIGURE 1. Different configurations based on the presence of overlapping regions among consecutive windows.**

to provide more powerful parallelism abstractions. Finally, we give an overview of the different models that can be used to design the run-time system (from now on RTS).

### A. WINDOWED QUERIES

Streaming applications extract relationships from continuous data flows in order to promptly enable decision-making activities. To deal with unbounded sequences of input items, such applications (queries in this domain) are often applied using a sort of stream discretization called *windowing* [9]. The idea is that the computation is continuously repeated on the most recent input items. As an example, suppose an application that processes a stream of transactions from a financial market. A query applied on temporal windows can be the following: "*generate a trade proposal when the spread between the average price of two stock symbols deviates by more than a given threshold in a period of five seconds*".

Different models have been presented in the literature [10]. The *triggering semantics* expresses when a window is ready to be processed while the *eviction semantics* states which are the data items that must be purged from the window after its triggering. Common semantics use the number of items (*count-based*) or a timestamp attribute in the items (*time-based*). As an example, time-based windows trigger every $s > 0$ time units (*sliding factor*) where each window spans over the last $w > 0$ time units (*window length*).

Three common configurations can be identified, as shown in Figure 1. *Tumbling windows* are fully disjoint and adjacent. *Sliding windows* are the most general case, with items that may belong to multiple consecutive windows. *Hopping windows* are again disjoint but not consecutive.

### B. EXISTING STREAMING INTERFACES

SPSs are frameworks for developing and running streaming applications. They provide a set of common transformations like map, flatmap, filter, and windowed operators which discretize the input stream and apply a user-defined function on each window. Window-based processing is often a time-consuming part that demands parallel execution [11]. In this section, we focus on how windowed operators are expressed in modern SPSs and how they can be accelerated.

The API of Apache Flink [2] is used to build pipelines of processing stages. It has a *fluent interface* written in Java

**Code listing**

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val events = env.addSource(new EventGenerator()).setParallelism(1)
val windowedCounts = events
        .assignTimestamps(new AdTimestampExtractor())
        .keyBy("event_type")
        .window(TumblingEventTimeWindows.of(1000))
        .fold(0) { (acc, ev) => acc = acc + 1 }.setParallelism(5)
windowedCounts.addSink(new TimestampingSink()).setParallelism(1)
env.execute("MyTest")
```

**LISTING 1.** Example of usage of the Apache Flink API.

**Code listing**

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("EventGenerator", new EventGenerator(), 1);
BaseWindowedBolt windowedCount = new CountWinBolt()
        .withTumblingWindow(new Duration(1, TimeUnit.SECONDS))
        .withTimestampField("event_time");
builder.setBolt("WindowAggregation", windowedCount, parallelism)
        .fieldsGrouping("EventGenerator", new Fields("event_type"));
builder.setBolt("Sink", new EventSink(), 1).shuffleGrouping("WindowAggregation");
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("MyTest", conf, builder.createTopology());
```

**LISTING 2.** Example of the Core Storm API in Java.

and Scala. Each `DataStream` object has methods to apply map, filter, reduce, join and windowed queries to derive new `DataStream` objects. Listing 1 exemplifies this usage. In the example, the application partitions the input records based on the event type attribute (`keyBy`), and counts the number of items in tumbling windows of one second.

The `fold()` function is used to update the count. Each computational phase has a parallelism (chosen with `setParallelism()`). Each replica of the windowed stage is executed by a different thread. In the example, Apache Flink assigns each different event type to one of five replicas.

Apache Storm [1] has a more verbose interface where each stage (*spout* for sources, *bolts* for the others) extends a proper class and overrides specific methods. Listing 2 shows the code instantiating the same toy application where `CountWinBolt` is a class extending the super class `BaseWindowedBolt` by overriding the method `execute()` that receives a collection of the items in the window and produces the corresponding result.

Parallelism is expressed by having more replicas of each stage (operator), and by specifying how the input items are distributed. The example uses a field grouping to distribute input items with the same key attribute to the same replica. By default, each replica is executed by a dedicated thread.

### C. TOWARDS PATTERN-BASED STREAM PROCESSING

We can identify three possible parallelism paradigms suitable to increase the throughput of windowed operators:

- *inter-key parallelism*: this is basically the most common parallelization provided in the existing systems, where replicas of the same stage process windows of different key groups in parallel;
- *inter-window parallelism*: consecutive windows of items even with the same key attribute can be assigned and processed in parallel by distinct replicas, e.g., the first window to the first replica, the second to the second replica and so forth;
- *intra-window parallelism*: when specific properties of the query function are known (e.g., associativity and commutativity), the processing of each window can be executed in parallel as a *map-reduce*, i.e. by splitting the window content and aggregating the partial results of the partitions into a window-wise result.

**TABLE 1.** Parallelism paradigms natively available to accelerate windowed operators.

| | Apache Flink | Apache Storm | Spark Streaming |
|---|---|---|---|
| Inter-key Parallelism | ✓ | ✓ | ✓ |
| Inter-window Parallelism | ✗ | ✗ | ✗ |
| Intra-window Parallelism | ✗ | ✗ | ✓ |

Not all of these paradigms are directly supported by the existing SPSs. For example, the third one is supported by Spark Streaming [12] (an SPS based on micro batching) while it is not primitively available in Apache Storm and Apache Flink. Table 1 summarizes which paradigms are supported in these SPSs.

In addition to provide a programming model where these three paradigms are immediately available to the high-level programmer, the approach proposed in this paper is aimed at providing a user-friendly programming model to compose and nest such paradigms in order to jointly exploit them. To do that, we adapt to the streaming domain the *parallel design patterns* methodology [8], [13]. Analogously to the design patterns used in object-oriented programming, a pattern is a high-level design, including possible implementation strategies and typical usages within different kinds of applications. In the years, the abstract parallel design pattern concept has converged into a real parallel programming methodology like in Intel TBB [14], FastFlow [7] and Microsoft PPL [15], where parallel patterns are directly made available to the users as abstractions of a sequential programming language (e.g., through objects or library calls).

Listing 3 shows an example in a generic object-oriented syntax. The structure of the application is defined in a bottom-up fashion, with two patterns created and nested. The programmer starts the execution of the application by running the topmost pattern (instance of the class `Pattern2`) which has two internal parallel entities which, in turn, are replicas of another pattern (class `Pattern1`). During the construction of each pattern, the user may provide to the constructors several configuration parameters and the business logic code (e.g., through lambda functions or functors).

Our goal is to define a new API for streaming pipelines adopting a similar pattern-based approach.

**Code listing**

```
// instantiate the innermost pattern (P1)
Pattern1 p1 = new Pattern1(arg1, arg2, ...);
p1.setParallelism(4);
// instantiate the outermost pattern (P2) passing P1 as an input parameter
Pattern2 p2 = new Pattern2(..., p1, ...);
p2.setParallelism(2);
// synchronous run of the topmost pattern
p2.run();
```

**LISTING 3. Example of pattern instantiation and execution.**

### D. DESIGN OF STREAMING RUN-TIME SYSTEMS

In addition to a user-friendly API, SPSs must adopt an efficient RTS targeting parallel hardware. Three models are suitable to implement the RTS:

- *thread-based parallelism*: each node of the data-flow graph (operator) is executed by a set of dedicated threads (one per replica of the operator). Furthermore, threads exchange data items through some sort of concurrent buffers/queues shared among threads;
- *actor-based parallelism*: to decouple the number of operators and their replicas from the number of threads, actor-based programming (like in Akka [16] and CAF [17]) can be used to design the RTS. Each replica is implemented by an independent actor, i.e. a computational entity that, in response to a message, executes a user-defined code and generates other messages to be sent to other actors. A pool of threads execute actors having at least one message ready in their input mailboxes (work-stealing strategies are often applied to balance the workload among threads);
- *task-based parallelism*: this model is widely diffused in the domain of High-Performance Computing in libraries such as Intel TBB [14] and its high-level interface FlowGraph. It shares some ideas of the actor-based programming, since tasks are lightweight execution entities like actors. However, here parallelism is expressed in a further additional dimension: each node in FlowGraph can spawn a potentially unlimited number of tasks which depends on the speed of messages arriving at the node and on the execution speed of its processing function.

Thread-based parallelism is the *de-facto* standard in existing SPSs available to the Big Data community (Apache Flink and Apache Storm adopt this model). One of the main reasons is that operators often perform blocking activities with external services (e.g., key-value stores, publish-subscribe and logging systems) which are not well suited for actor-based and task-based parallelism approaches. However, such models (notably task-based parallelism) are very powerful to mitigate locality issues and to provide transparent load balancing, although the task granularity must be carefully chosen to obtain good performance.

Although our patterns and the high-level interface are independent of the model adopted by the RTS, we present an implementation with the building blocks available in the FastFlow library, which is based on thread-based parallelism. The software stacks of this solution are shown in Figure 2. We will study in the future how to exploit alternative RTSs for the implementation of our patterns, although a preliminary study is presented in the concluding part of the paper.

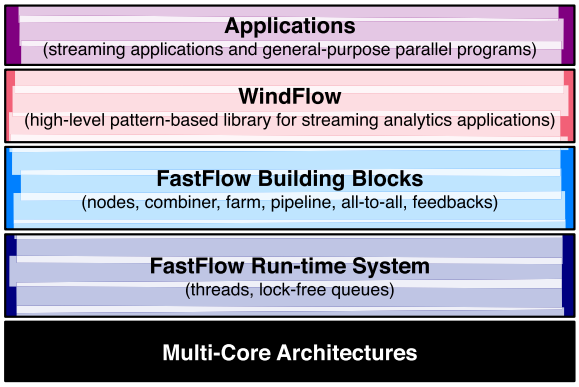


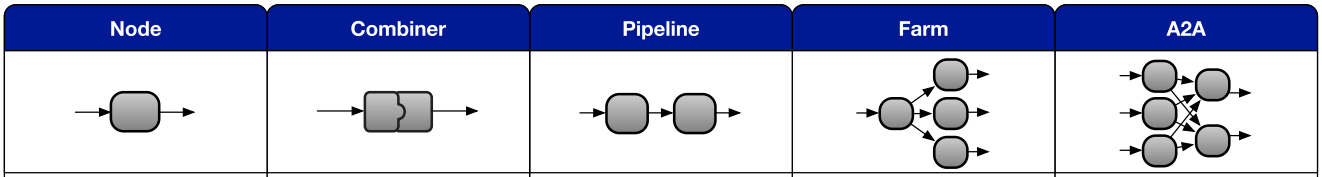

**FIGURE 2. Layered software architecture of the FastFlow-based implementation of the WindFlow library.**

## III. FASTFLOW OVERVIEW

FastFlow is a library in C++ targeting multi/many-core platforms [7]. Starting from version 3.0, the library offers a two-level API. The first is used by the high-level programmer to develop parallel programs, while the second provides a set of low-level *building blocks* [18] for developing new RTSs for domain specific languages and libraries.

The FastFlow library is realized as a modern C++ header-only template library that allows the programmer to develop parallel applications modeled as a directed data-flow graph of processing *nodes*. Following the thread-based parallelism model, each FastFlow node represents a sequential computation component executed by a dedicated thread. Each node can have zero, one or more input channels and zero, one or more output channels. The graph of concurrent/parallel nodes is constructed by the assembly of sequential and parallel building blocks. Input and output communication channels are implemented through Single-Producer Single-Consumer (SPSC) FIFO queues. Operations on such queues (that can have either bounded or unbounded capacity) are based on non-blocking lock-free synchronizations enabling fast data processing in high-frequency streaming applications [19].

FastFlow channels do not carry plain data but references to heap-allocated data. The semantics of sending data references over a communication channel is that of transferring the ownership of the data pointed by the reference from the sender node (producer) to the receiver node (consumer) according to the *producer-consumer* model. The data reference is *de facto* a *capability*, i.e. a logical token that grants access to a given data or to a portion of a larger data structure. Based on this *reference-passing* semantics, the receiver is expected to have exclusive access to the data reference received from one of the input channels, while the producer is expected not to use the reference anymore.




```
class MyNode:
 public ff_node_t<T1, T2>
{
 ...
 T2 *svc(T1 *input)
 {
   ... // business logic
 }
};
...
int main()
{ ...
 MyNode node(...);
 node.run_and_wait_end();
}
```

```
class MyNode1:
 public ff_node_t<T1, T2> {...};
class MyNode2:
 public ff_node_t<T2, T3> {...};
...
int main()
{ ...
 MyNode1 n1(...);
 MyNode2 n2(...);
 ff_comb comb(n1, n2);
 comb.run_and_wait_end();
}
```

```
class MyNode1:
 public ff_node_t<T1, T2> {...};
class MyNode2:
 public ff_node_t<T2, T3> {...};
...
int main()
{ ...
 MyNode1 n1(...);
 MyNode2 n2(...);
 ff_Pipe<T1,T3> pipe(n1, n2);
 pipe.run_and_wait_end();
}
```

```
class Emitter:
 public ff_monode_t<T1, T1> {…};
class Worker:
 public ff_node_t<T1, T2> {...};
...
int main()
{ ...
 Emitter e(...);
 Worker w(...);
 vector<Worker> ws(3, w);
 ff_Farm<T1,T2> farm(w);
 farm.add_emitter(e);
 farm.run_and_wait_end();
}
```

```
class NodeLeft:
 public ff_monode_t<T1, T2> {…};
class NodeRight:
 public ff_minode_t<T2, T3> {…};
int main()
{ ...
 NodeLeft nl(...);
 NodeRight nr(...);
 vector<NodeLeft> wl(3, nl);
 Vector<NodeRight> wr(2, nr);
 ff_a2a a2a();
 a2a.add_firstset(wl);
 a2a.add_secondset(wr);
 a2a.run_and_wait_end();
}
```

**FIGURE 3.** Basic building blocks available in FastFlow version `3.0`.

In the following, we briefly describe the basic building blocks available in the FastFlow library version 3.0. The API of each building block is sketched in Figure 3.

Node The *node* is the basic abstraction of the building blocks. It defines the unit of sequential execution in the FastFlow library. A node encapsulates either user's code (i.e. business logic) or RTS code. User's code can also be wrapped by a FastFlow node executing RTS code to manipulate and filter input and output data before and after the execution of the business logic code. Based on the number of input/output channels it is possible to distinguish three different kinds of sequential nodes: *standard* with one input and one output channel, *multi-input* with many inputs and one output channel, and finally *multi-output* with one input and many outputs. A generic node performs a loop that: *i)* gets a data item (through a memory reference to a data structure) from one of its input queues; *ii)* executes a functional code working on the data item and possibly on a state maintained by the node itself by calling its service method `svc()`; *iii)* puts a memory reference to the resulting item(s) into one or multiple output queues selected according to a predefined or user-defined policy.

Combiner The *combiner* building block allows the user to combine two nodes into one single sequential node. Conceptually, the operation of combining sequential nodes is similar to the composition of two functions. In this case, the functions are the service functions of the two nodes (e.g., the `svc()` method). This building block promotes code reuse through fusion of already implemented nodes and it can also be used to reduce the threads used to run the data-flow network by executing the functions of multiple nodes by a single thread.

Pipeline The *pipeline* allows building blocks to be connected in a linear chain. It is used both as a container of building blocks as well as an application topology builder. At execution time, the *pipeline* building block models the data-flow execution of its building blocks on data elements flowing in a streamed fashion.

Farm The *farm* building block models functional replication of building blocks coordinated by a master node called *Emitter*. The simplest form is composed of two computing entities executed in parallel: a multi-output master node (the Emitter), and a pool of pipeline building blocks called *Workers*. The Emitter node schedules the data elements received in input to the Workers using either a default policy (i.e. *round-robin* or *on-demand*) or according to the algorithm implemented by the user code defined in its service method. In this second scenario, the stream elements scheduling is controlled by the user through a custom policy.

A2A The *All-to-All* (briefly *A2A*) building block defines two distinct sets of *Worker*s connected as in a full crossbar. This means that each Worker in the first set (called L-Worker) is connected to all the Workers in the second set (called R-Workers). Although the topological shape is a full crossbar, the user can implement any custom distribution in the L-Workers (e.g., sending each data item to a specific R-Worker, shuffling o broadcasting).

Based on these building blocks, we describe in the rest of this paper an implementation of the RTS for our WindFlow parallel library.

## IV. WINDFLOW PARALLEL PATTERNS

Following the pattern-based vision outlined in the previous sections, in this paper we introduce WindFlow[2]. The library provides four parallel patterns that can be used to implement high-throughput and low-latency sliding-window queries. Figure 4 depicts with an UML class diagram the relationship among our patterns and the FastFlow building

[2]https://github.com/ParaGroup/WindFlow

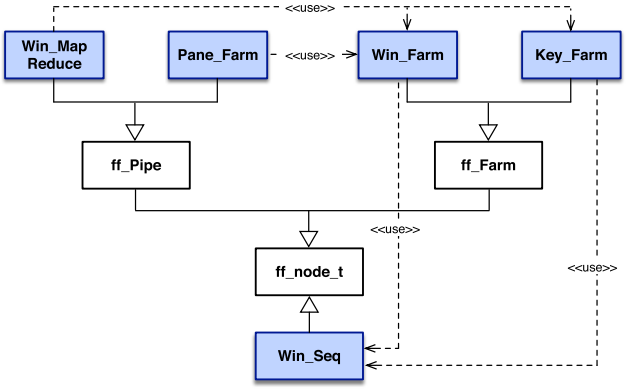



**FIGURE 4.** UML diagram of the sliding-window parallel patterns provided by the library and their relationship with the FastFlow building blocks.

**TABLE 2.** Parallel paradigms provided by each pattern.

| | Windowed Farm | Keyed Farm | Paned Farm | Windowed Map-Reduce |
|---|---|---|---|---|
| Inter-key Parallelism | ✗ | ✓ | ✗ | ✗ |
| Inter-window Parallelism | ✓ | ✗ | ✓ | ✗ |
| Intra-window Parallelism | ✗ | ✗ | ✓ | ✓ |

blocks. The four parallel patterns, `Win_Farm`, `Key_Farm`, `Pane_Farm` and `Win_MapReduce`, are classes extending either the `ff_Pipe` or the `ff_Farm` FastFlow classes.

As Table 2 shows, the patterns are used to express parallelism among different windows, within each window or between logical substreams. Although some patterns already exploit more than one parallelism paradigm, patterns nesting makes it possible to combine multiple paradigms in a general manner within a single application structure. This feature will be discussed in Section V.

The library supports the most common windowing models and query definitions:

- *count-based* and *time-based* windows. The window length and sliding factor are expressed in number of input items or in time units. All the configurations are admissible (i.e. tumbling, sliding and hopping);
- the query logic is instantiated using either a *non-incremental* or an *incremental* interface. In the first case, the user-defined function is called every time a new window is complete by providing access to the window items through iterators. In the second case, for each new input item the RTS calls the function several times to update the results of the windows including the item in their scope. The two signatures are both accepted as input of the pattern constructor via overloading.

In the next part, we will describe for each pattern its API and how it has been implemented in the library.

### A. SEQUENTIAL PATTERN

The sequential pattern is implemented by the `Win_Seq` class. During the pattern execution, windows are processed

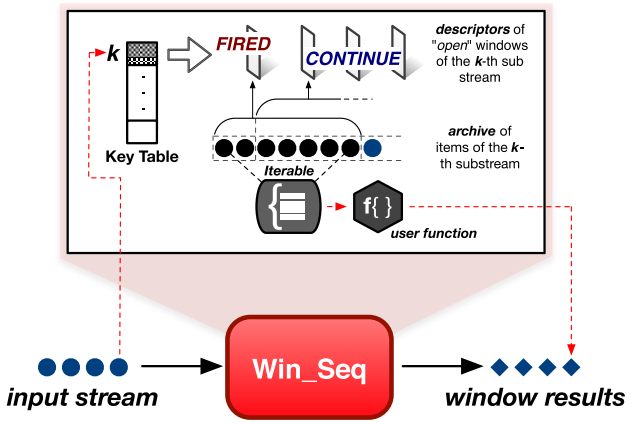



**FIGURE 5.** `Win_Seq` supports non-keyed/ keyed streams with non-/incremental count-/time-based windows.

sequentially and in order. Figure 5 shows the conceptual structure of the pattern. The pattern is configured to work with keyed streams where input items are grouped by their key identifier and an output result is produced for each complete window of items with the same key attribute.

#### 1) IMPLEMENTATION

The pattern is implemented as a C++ template class extending the base class `ff_node_t` of single-input single-output FastFlow nodes. The pattern maintains a hashmap `KeyTable` (shortly, `KT`) indexed by the key attribute. Each entry of the table stores two fields:

- a set of *descriptors* of *open* windows, i.e. not-complete windows for which we have received at least one item. Each descriptor provides an `onEvent()` method, called with the current input item, which returns whether the window is complete or not by comparing the item identifier/timestamp with the window ending boundary. The method returns two possible outcomes: `CONTINUE` if the window is not complete, or `FIRED` otherwise. The descriptor stores the window result that is filled by the query function processing;
- an *archive* containing all the items belonging to windows that are not complete yet. Such items are kept ordered for an efficient lookup into the archive.

For each received item, the RTS executes the `processItem()` function described in Algorithm 1. The routine identifies all the windows containing the item, creates the descriptors, and calls the `onEvent()` method on each of them. If a non-incremental definition is used, the query function is called for each window that returns the `FIRED` outcome. The function has access to the window result (a user-defined data structure allocated by the RTS) and to all the items of the window through a custom `Iterable<T>` object, with `T` the type of the input item. The `Iterable` object provides, without any copy, a *logical read-only view* of the archive limited to the items in the window. The object can be used as a STL container.

**Algorithm 1** Processing of Input Items by `Win_Seq`

```
1:  procedure processItem(t)
2:      (key, id, ts) ← t.getControlFields()
3:      (W, archive) ← KT[key]
4:      archive.insert(t)
5:      for each win ∈ W containing t do
6:          outcome ← win.onEvent(t)
7:          if outcome == FIRED then
8:              if NIC then
9:                  (first, last) ← archive.findIterators(win)
10:                 Iterable<T> view(first, last)
11:                 call winFunction(win.wid, view,
    win.result)
12:             send win.result to the output queue
13:             archive.purge(win.start, slide)
14:         if outcome == CONTINUE and ¬NIC then
15:             call winFunction(win.wid, t, win.result)
16: end
```

If the query function is provided with the incremental interface, it is called for each window that produces a `CONTINUE` outcome by passing the new item and the window result to be updated as a reference. In the pseudocode, the flag `NIC` is true if the query function is compliant with the non-incremental signature. After the processing of a fired window, its descriptor is removed and all the items no longer needed by any of the open windows are purged from the archive (line 13). The archive in case of incremental queries is optional as the access to historical data is not needed.

### 2) HIGH-LEVEL API

To ease the creation of the pattern, the library provides a builder class with a *fluent interface*. Using the builder, the configuration parameters can be provided to the pattern constructor in any order with default values if not specified. Listing 4 shows an example of instantiation with a query that simply counts the items in the window. The application is a pipeline of three stages: the first is in charge of generating the stream (by reading the items from an input file), the second is the `Win_Seq` instance, and the third stage receives the results and stores them in a file. In the code, `tuple_t` and `result_t` are the input item and the result data types.

In the code snippet, the query is instantiated with the non-incremental signature and created as a lambda function. The builder's method `withCBWindow()` is used to set the window length and slide parameters while `withName()` assigns to the pattern instance a name as a string. Then, the pattern object is created by invoking the `build()` method of the builder. Thanks to the CTAD feature recently introduced in C++17 (Class Template Argument Deduction), the template arguments of the `Win_Seq` class (`tuple_t` and `result_t`) are deduced by the compiler from the function type passed as parameter to the builder constructor.

**Code listing**

```
...
auto countF = [] (size_t wid, Iterable<tuple_t> &win, output_t &r) {
   r.value = win.size();
}
Generator gen(input_filename);
Consumer cons(output_filename);
// creation of the Win_Seq pattern
Win_Seq seq = WinSeq_Builder(countF).withCBWindow(w, s)
                                    .withName("my_seq")
                                    .build();
ff_Pipe pipe(gen, seq, cons);
pipe.run_and_wait();
...
```

**LISTING 4.** Example of instantiation of the `Win_Seq` pattern.

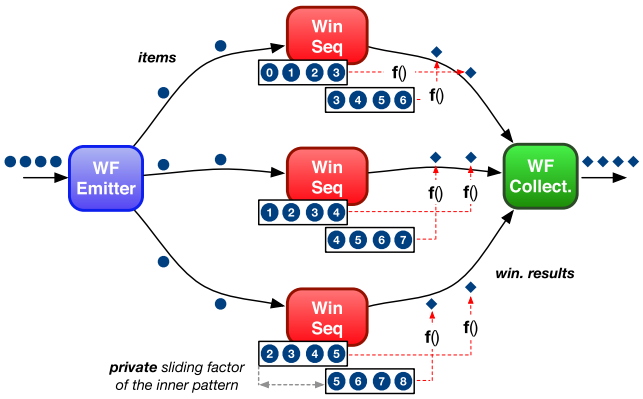



**FIGURE 6.** `Win_Farm` used with three `Win_Seq` internal instances. In this example, the `Win_Farm` works with count-based windows with $w = 4$ and $s = 1$ items. Each `Win_Seq` instance has a private sliding factor of $3 \cdot s$ items.

After the instantiation, the pattern is added as the second stage of a FastFlow pipeline, together with the `Generator` and the `Consumer` nodes which extend the `ff_node_t` base class of FastFlow. Then, the pipeline is executed synchronously with respect to the main thread until the stream has been entirely processed.

## B. WINDOWED FARM

The `Win_Farm` pattern enables inter-window parallelism. It replicates an inner pattern according to the *replication degree* $n > 0$ chosen by the user. The internal instances work in parallel on distinct windows. The behavior is shown in Figure 6 with the pattern having a pool of `Win_Seq` instances inside, each using the same windowing model (count- or time-based) and window length/sliding factor.

The pattern has two support entities implemented as FastFlow nodes and executed by dedicated threads of the RTS. The `WF_Emitter` distributes the items such that each replica receives all the items needed to compute the assigned windows. The `WF_Collector` performs the collection task that restores the order of results.

### 1) IMPLEMENTATION

The `Win_Farm` class extends the `ff_Farm` class of FastFlow by replacing the default emitter and collector

with instances of the `WF_Emitter` and `WF_Collector` classes. The two important points of the implementation are related to the windows assignment and the distribution of input items received from the stream. The assignment of windows to the internal instances is performed in a circular manner. The key attribute type `key_t` of the input items must be hashable, i.e. `std::hash<key_t>()` must be defined. Let $k$ be the hashcode of the key value, the processing of the $i$-th window of that key is assigned to the $j$-th internal pattern such that:

$$j = [(k \mod n) + i] \mod n \tag{1}$$

This choice guarantees that the first internal instance of the round-robin assignment is not the same for all the keys.

The replicas within the pattern work in parallel on different windows. The figure assumes that all the items have the same key attribute to show that this pattern does not need keyed streams. Each replica is in charge of computing a subset of the windows: the first instance processes windows $W_0, W_3, W_6 \ldots$, the second $W_1, W_4, W_7 \ldots$, and so forth. To do that, each replica receives only the items belonging to at least one of its assigned windows, while each item can be sent to more than one replica.

The pattern needs to determine for each input which are the windows "touched" by that item. The mapping is calculated using the parameters $(w, s)$ of the query and the identifier/timestamp of the item. The distribution logic is described in Algorithm 2 for count-based windows. The time-based case is analogous, with the timestamp attribute involved in the expression at line 7 and 9.

**Algorithm 2** Distribution of Input Items

```
1:  procedure distributeItem(t)                ▷ t is a new input item
2:      (key, id, ts) ← t.getControlFields()
3:      k ← std::hash<key_t>()(key)
4:      if id < w then
5:          first ← 0
6:      else
7:          first ← ⌈(id + 1 − w)/s⌉
8:      last ← ⌈(id + 1)/s⌉ − 1
9:      D_dst ← ∅
10:     for i ← first to last do
11:         D_dst ← D_dst ∪ {((key mod n) + i mod n}
12:     for each j ∈ D_dst do
13:         send t to the j-th internal instance
14: end
```

The pseudocode computes the range of windows $[first..last]$. For sliding windows with a small sliding factor, the same item is in general present in many consecutive windows, by requiring a high communication overhead spent by the emitter node. However, we point out that this is mitigated in FastFlow, where data messaging is implemented by exchanging memory pointers.

**Code listing**

```
...
Generator gen(input_filename);
Consumer cons(output_filename);
// creation of the Win_Farm pattern
Win_Farm wf = WinFarm_Builder(countF).withCBWindow(w, s)
                                     .withName("my_wf")
                                     .withParallelism(parallelism)
                                     .withOrdering(true)
                                     .build();
ff_Pipe pipe(gen, wf, cons);
pipe.run_and_wait();
...
```

**LISTING 5.** **Example of instantiation of the `Win_Farm` pattern.**

#### 2) HIGH-LEVEL API

The pattern has a builder class as the `Win_Seq`. The replication degree is configured by calling `withParallelism()`, while the behavior of the collector node, that is whether it must order the results with the same key is set through `withOrdering()`. Listing 5 shows an example of instantiation using the same query of Listing 4.

### C. PANED FARM

The `Pane_Farm` pattern is an parallelization of the approach in [20]. The approach avoids recomputing windows from scratch by exploiting the partial overlapping between two consecutive windows. It is based on a two-level aggregation that uses the notion of *pane*: panes are tumbling windows of length equal to $p = GCD(w, s)$. The length is in number of items or in time units depending on the windowing model. The computation consists of two phases: the *Pane-level Sub-Query* (PLQ) computes a result for each pane, while in the *Window-level Sub-Query* (WLQ) the results of the $w/p$ panes belonging to the same window are merged to produce the corresponding window result.

The `Pane_Farm` pattern is a pipeline of two stages (PLQ and WLQ), see Figure 7. The figure shows three alternative cases, with the second stage parallel and the first sequential, the opposite, and with both stages parallel.

#### 1) IMPLEMENTATION

The pattern is provided as a class extending the `ff_Pipe` of FastFlow. The pipeline is built with two stages that are instances either of the `Win_Seq` or of the `Win_Farm` pattern according to the replication degree chosen. The first stage works with count- or time-based windows depending on the query definition. The second stage works with count-based windows of length $w/p$ that slide every $s/p$ pane results. This pattern is a first example of a layered design, where complex patterns are implemented as combination of other simpler patterns.

#### 2) HIGH-LEVEL API

The builder class of the `Pane_Farm` allows the user to easily configure the pattern, the ordering behavior out of the last stage and the replication degrees. The builder constructor

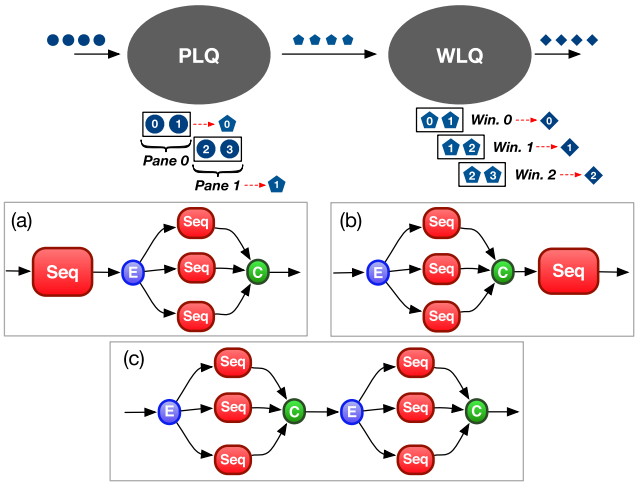



**FIGURE 7.** `Pane_Farm` **pattern working with count-based windows with** $w = 4$ **and** $s = 2$ **items.**

**Code listing**

```
...
auto countF = [] (size_t pid, Iterable<tuple_t> &win, output_t &r) {
    for (auto t: win)
        if (t.value == 10) r.count++;
}
auto sumF = [] (size_t wid, const output_t &p, output_t &r) {
    r.count += p.count;
}
Generator gen(input_filename);
Consumer cons(output_filename);
// creation of the Pane_Farm pattern
Pane_Farm pf = PaneFarm_Builder(countF, sumF).withCBWindow(w, s)
                                             .withName("my_pf")
                                             .withParallelism(par1, par2)
                                             .withOrdering(true)
                                             .build();
ff_Pipe pipe(gen, pf, cons);
pipe.run_and_wait();
...
```

**LISTING 6.** **Example of instantiation of the** `Pane_Farm` **pattern.**

receives two functions, the first to compute pane results, the second to aggregate them to produce window-wise results. Listing 6 shows an example of instantiation. The query counts the occurrences of the integer 10 in the window. In the example, the PLQ function has a non-incremental signature while the WLQ function has an incremental definition.

### D. KEYED FARM

The `Key_Farm` pattern expresses inter-key parallelism. To scale with the number of replicas, the pattern needs a keyed stream with a large number of distinct values of the key attribute. Figure 8 shows the pattern instantiated with count-based windows having $w = 4$ and $s = 1$. In this example, the input stream conveys items belonging to three logical substreams assigned to three internal instances.

#### 1) IMPLEMENTATION

The `Key_Farm` class extends `ff_Farm`. Each instance of the inner pattern works with exactly the same windowing

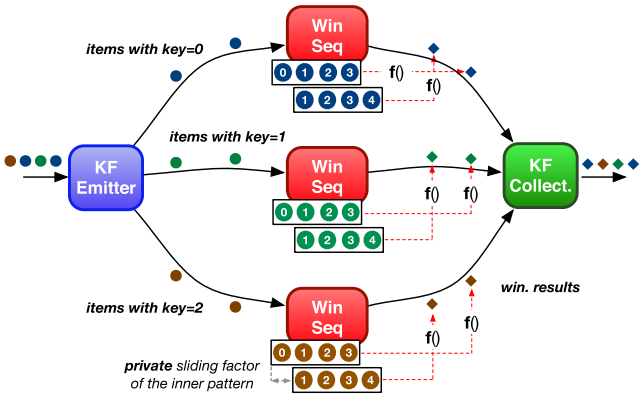



**FIGURE 8.** `Key_Farm` **pattern with count-based windows,** $w = 4$ **and** $s = 1$**, and three substreams assigned to three** `Win_Seq` **internal instances.**

specification of the original query (in the example using $w = 4$ and $s = 1$). The distribution and collector nodes instantiate the `KF_Emitter` and `KF_Collector` classes. The first uses a hash function to map the key values onto the internal patterns. Since windows of the same substream are processed by the same internal instance, they are produced in increasing order. Therefore, the collection task is straightforward.

#### 2) HIGH-LEVEL API

The API provides a builder class `KeyFarm_Builder` to instantiate this pattern analogously as we did for the `Win_Farm` pattern in Section IV-B. The builder provides the same methods to specify the pattern name, the window type, length and slide, and the replication degree.

### E. WINDOWED MAP-REDUCE

The last pattern `Win_MapReduce` expresses intra-window parallelism. The idea is to split each window in partitions (subsets of items), compute partial results one per partition in the *map* phase, and use them to assemble window-wise results in the *reduce* stage.

Although the idea is similar to the one behind the `Pane_Farm` pattern, this pattern uses a different and more general notion of partition. A pane consists of items with consecutive identifiers/timestamps and within a range of a fixed size independent of the replication degree. In the `Win_MapReduce` pattern instead, items are assigned to the partitions in a circular manner. Figure 9 shows an example with count-based windows with $w = 6$ and $s = 2$, where each window is split into two partitions (in general, the number of partitions is equal to the replication degree of the *map*). In the example, the `Win_MapReduce` pattern assigns three items to each partition, while the `Pane_Farm` pattern would be forced to use panes of two items long. In this way, the `Win_MapReduce` enables intra-window parallelism also for queries using small sliding factors or even when windows are tumbling or hopping, where the paned approach does not have sense. However, consecutive win-

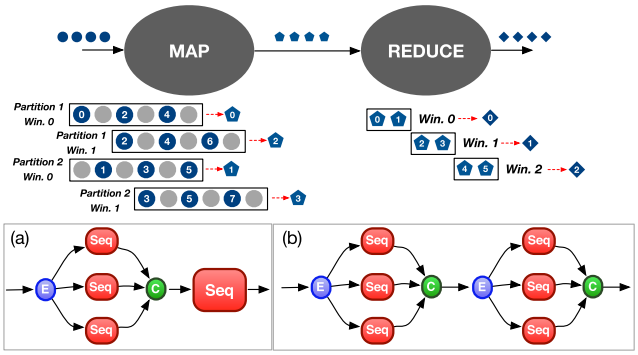



**FIGURE 9.** `Win_MapReduce` **pattern: example with count-based windows with** $w = 6$ **and** $s = 2$ **items.**

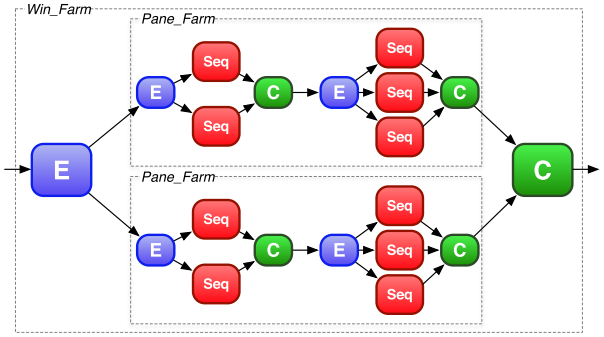



**FIGURE 10.** **Example of a complex nested structure: a** `Pane_Farm` **instance is replicated two times inside a** `Win_Farm` **pattern.**

dows are always computed from scratch although this happens in parallel.

#### 1) IMPLEMENTATION

The pattern is implemented as a pipeline of a `Key_Farm` instance followed by a `Win_Farm` instance (or a `Win_Seq` if the *reduce* phase is sequential). The RTS uses the `Key_Farm` pattern to implement the *map* phase by passing a proper routing function to its constructor. While in the standard case the pattern expects a hash function that uniquely maps key attributes onto the internal replicas, the RTS builds this `Key_Farm` instance by passing a function that distributes the items in a circular manner regardless their key attribute. Although in general this is not the correct use of the `Key_Farm` pattern, this implements the map behavior: each window materializes in all the internal replicas, each actually receiving a subset of the expected items.

In Figure 9, we assume to have two `Win_Seq` instances in the `Key_Farm`: items $0, 2, 4$ are distributed to the first `Win_Seq` instance while $1, 3, 5$ are directed to the second one. The triggering semantics is based on the comparison between the items' identifier (or timestamp) and the ending boundary of the windows. In this example, as soon as the `Win_Seq` instances receive the first item with identifier greater than 5, the corresponding partition of the first window is closed and its partial result produced to the *reduce* stage. This happens at the arrival of item 6 in the first `Win_Seq` instance, while the second one considers its partition of the first window complete after the arrival of item 7.

The second phase applies the reduce function to the results of the partitions produced by the *map* stage. Each window has a fixed number of partitions and the reduce phase can be modeled as a tumbling-window computation with $w = s = n$, where $n > 0$ is the replication degree used in the *map* stage. If the *reduce* phase is lightweight, it can be executed by a `Win_Seq` instance as depicted in Figure 9(a). Otherwise, it can be parallelized by a `Win_Farm` as in Figure 9(b).

#### 2) HIGH-LEVEL API

The builder of this pattern (`WinMapReduce_Builder`) has the same identical usage of the `Pane_Farm`'s one. Also in this case, the *map* and *reduce* functions can be provided using either the non-incremental/incremental signatures.

## V. NESTING OF PATTERNS AND TRANSFORMATIONS

In this section we study two important features of the library: *i)* the possibility to build complex nested structures of patterns following the idea sketched in Listing 3; *ii)* the library provides different transformations that reshape the pattern implementation by merging/replicating specific functionalities to improve performance or increase efficiency.

### A. COMPLEX NESTING OF PATTERNS

In their default usage, the `Win_Farm` and `Key_Farm` patterns have internal instances that replicate the `Win_Seq` pattern. A powerful feature is to allow the replication of parallel patterns, notably the `Pane_Farm` and `Win_MapReduce`. Figure 10 shows an example of such an idea. The `Pane_Farm` pattern is instantiated using a replication degree of 2 and 3 in the PLQ and WLQ stages respectively, while the whole `Pane_Farm` instance is replicated two times within a topmost `Win_Farm` pattern.

Table 3 summarizes for each admissible complex nested structure why it could be effective from the performance viewpoint. In particular:

- *increase the pane length*: when the `Pane_Farm` can be applied, it represents an effective choice because it avoids recomputing each window from scratch. However, if the panes are too small such advantage is sharply reduced. In the `Win_Farm` pattern, each internal instance uses a private sliding factor $n-$times greater than the original one (as in Figure 6), with $n > 0$ the replication degree. Therefore, the nesting of `Win_Farm` with a `Pane_Farm` can be used to take advantage of using larger panes;
- *alleviate the emitter overhead*: the emitter of the `Win_Farm` may multicast each item to a large set of destinations. Although only a memory pointer is forwarded in FastFlow, such distribution overhead may be significant with many replicas. By replicating a parallel pattern within the `Win_Farm` instance, we reduce its

**TABLE 3. Reasons for using complex nested structures.**

| Nesting | Reason |
|---|---|
| `Win_Farm(Pane_Farm)` | increase the pane length |
| `Win_Farm(Win_MapReduce)` | alleviate the overhead of the `Win_Farm` emitter |
| `Key_Farm(Pane_Farm)` `Key_Farm(Win_MapReduce)` | find more parallelism in case of few distinct keys |

**Code listing**

```
...
Generator gen(input_filename);
Consumer cons(output_filename);
// creation of the Pane_Farm pattern
Pane_Farm pf = PaneFarm_Builder(countF, sumF).withCBWindow(w, s)
                                             .withName("my_pf")
                                             .withParallelism(2, 3)
                                             .build();
// creation of the Win_Farm pattern
Win_Farm wf = WinFarm_Builder(pf).withName("my_wf")
                                 .withParallelism(2)
                                 .withOrdering(true)
                                 .build();
ff_Pipe pipe(gen, wf, cons);
pipe.run_and_wait();
...
```

**LISTING 7. Example of instantiation of the complex nested structure shown in Figure 10.**

replication degree and thus the communication overhead spent by its emitter functionality;

- *shortage of distinct keys*: when the stream conveys items with a key attribute assuming few distinct values, the parallelism exploitable by the `Key_Farm` pattern is limited as well as its scalability. This can be mitigated by using parallel patterns expressing inter-/intra-window parallelism within the `Key_Farm`.

Such nested structures are developed in a bottom-up fashion in the library. This is shown in Listing 7 where the `Win_Farm` pattern is built by passing to the constructor of its builder the instance of the pattern to be replicated (a `Pane_Farm` instance in the example).

The definition of the `WinFarm_Builder` constructor is overloaded. Instead of the query function to be used by the `Win_Seq` instances, the pattern to be replicated is provided. The library builds the streaming graph of Figure 10 where each internal instance is configured to use the same window length of the topmost pattern with a sliding factor that is $n$-times the original one.

### B. TRANSFORMATIONS

Complex structures like the one in Figure 10 should be modified before being executed. In particular, in the thread-based parallelism model adopted by the FastFlow RTS the replication of patterns and their composition leads to the presence of several distribution and collection entities implemented by dedicated threads. When such functionalities along a pipeline are underloaded, it may be useful to merge them to save

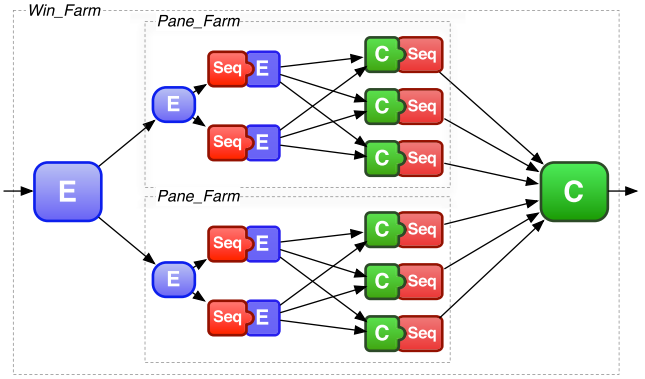



**FIGURE 11. Example of transformation of the FastFlow graph previously shown in Fig. 10.**

communication overhead. This concept is an application of the *operator fusion* or *chaining* technique [21].

The library provides an approach to restructure the graph obtained by nesting patterns. The pattern construction is enhanced with an additional parameter called *transformation level*. We introduce three levels with different effects on the whole structure. As an example, Figure 11 represents a reshaped version of the structure in Figure 10, where the `Pane_Farm` pattern is created with a transformation level equal to `LEVEL2`, which has the following consequences:

- the PLQ stage consists of a set of identical FastFlow nodes, each one obtained as a sequential composition (see Sect. III) of the original `Win_Seq` instance of the PLQ stage of Figure 10 combined with a copy of the emitter of the WLQ stage. In this way, each instance is in charge of receiving input items, computing the results of the assigned panes which are directly transmitted to the WLQ stage;
- the WLQ stage in turn has a set of identical FastFlow nodes, each is a sequential composition of a collector functionality of the PLQ and the original `Win_Seq` instance of the WLQ stage in Figure 10. Each collector buffers the received pane results by restoring their order. Then, the `Win_Seq` instances in the WLQ are in charge of aggregating pane results of the same windows to produce window results.

By referring to FastFlow (Sect. III), the structure within each each `Pane_Farm` is now a `ff_a2a` building block.

Once created with that transformation level, the `Pane_Farm` object is used to build the outermost `Win_Farm` instance. Again, the transformations are applied in a bottom-up fashion by passing to the `WinFarm_Builder` a new transformation level used by the topmost pattern. In the example, we use a level equal to `LEVEL1` which corresponds to removing the collector nodes of the WLQ stages by doing the whole ordering in the last `Win_Farm` collector node. The code to build the structure is shown in Listing 8, and uses the `withTransformation()` method of the builder.

Figure 12 shows all the transformations available for building the `Pane_Farm` and the `Win_MapReduce` patterns.

**Code listing**

```
...
Generator gen(input_filename);
Consumer cons(output_filename);
// creation of the Pane_Farm pattern (Transformation Level = 2)
Pane_Farm pf = PaneFarm_Builder(countF, sumF).withCBWindow(w, s)
                                             .withName("my_pf")
                                             .withParallelism(2, 3)
                                             .withTransformation(LEVEL2)
                                             .build();
// creation of the Win_Farm pattern (Transformation Level = 1)
Win_Farm out_wf = WinFarm_Builder(pf).withName("my_wf")
                                     .withParallelism(2)
                                     .withOrdering(true)
                                     . withTransformation(LEVEL1)
                                     .build();
ff_Pipe pipe(gen, out_wf, cons);
pipe.run_and_wait();
...
```

**LISTING 8. Instantiation of the complex nested structure with transformations (highlighted in the code).**

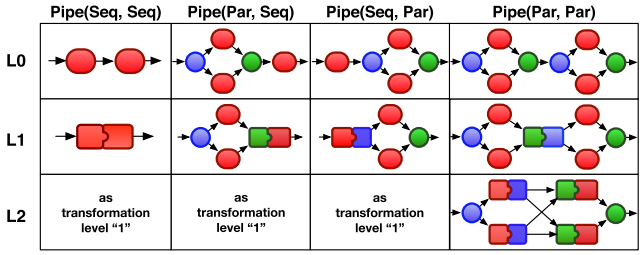



**FIGURE 12. Complete set of transformations supported for the `Pane_Farm` and the `Win_MapReduce` patterns.**

The rows are the transformation levels and the columns correspond to specific configurations according to the replication degree used for the first stage (PLQ or MAP) and for the second stage (WLQ or REDUCE). The notation `seq` means sequential, while `par` means parallel. The term `pipe` denotes the pipeline.

Figure 13 shows instead the effect of the transformation levels on the topmost pattern (`Win_Farm` or `Key_Farm`). Besides the first level discussed before, which removes additional collector nodes, the second transformation level has effects when the first stage of the internal instances (i.e. the PLQ or MAP stage) is parallel. In that case, the emitter functionalities are combined into a single custom node performing the hierarchical distribution.

We point out that such transformations are necessary to deal with the choice of mapping one node onto a dedicated thread. Although this solution is also common to the other existing SPSs, the RTS could be developed with other parallelism models in mind, preventing such issues but introducing other aspects to take care of. This aspect will be recalled in the last part of the paper.

## VI. EXPERIMENTS

In this section, we propose a detailed experimental evaluation. The goal is twofold: first to assess the effectiveness of the patterned abstractions and of their complex composition/nesting

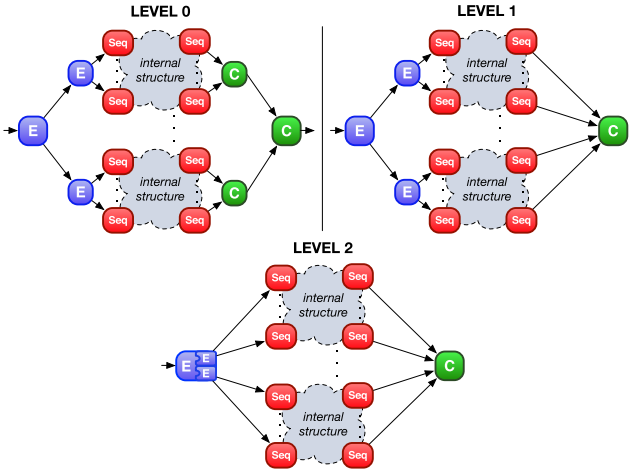



**FIGURE 13. Transformations of the `Win_Farm` and `Key_Farm` patterns when they replicate internal complex instances (i.e. `Pane_Farm` and `Win_MapReduce`).**

to increase performance; second to evaluate the efficiency of our RTS based on FastFlow and targeting shared-memory systems compared with existing JVM-based SPSs.

All the experiments have been developed using the WindFlow library (version `1.2`) and FastFlow (version `3.0`) on an Intel Phi Knights Landing (KNL) architecture equipped with 64 cores working at 1.3 GHz supporting four logical thread contexts per core. The machine is configured with 96 GB of DDR4 RAM and run Linux 3.10.0 × 86_64 shipped with Centos `7.2`. The compiler used is `gcc` version `7.3` with the `-O3` optimization flag.

### A. DESIGN SPACE EXPLORATION

Among the possible compositions, there is in general a small set of candidates that are suitable for the query parallelization. This depends on the properties of the function (i.e. whether it allows a decomposable computation) and on the windowing parameters, in particular how long is the sliding factor and its proportion with the window length. A *Design Space Exploration* (DSE) procedure is shown in Figure 14. Conceptually, it should be keep in mind by any programmer. It consists in a set of alternative choices:

- *does the stream convey items belonging to independent groups (keys)?* A positive answer justifies the use of the `Key_Farm` pattern and of its compositions with other patterns. The answer depends on the number of keys and their frequency and distribution skewness;
- *is the query decomposable?* This depends on how the window results are computed from the input data. While `Key_Farm` and `Win_Farm` are totally generic patterns, `Pane_Farm` and `Win_MapReduce` need that each result can be computed by aggregating results of window partitions, which is not always possible;
- *is the windowing model a sliding one?* The choice of the `Pane_Farm` pattern, which avoids recomputing consecutive windows from scratch, is constrained by the use

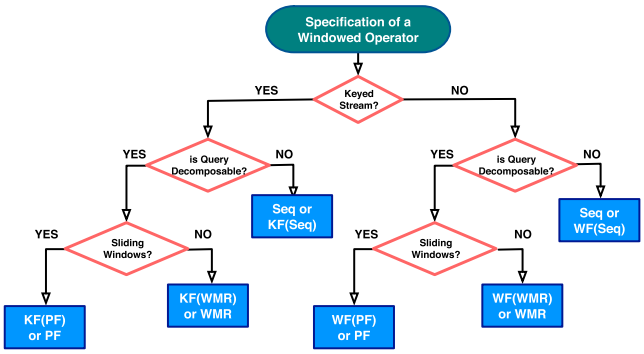



**FIGURE 14.** Design space exploration phase to use the parallel patterns provided in the library.

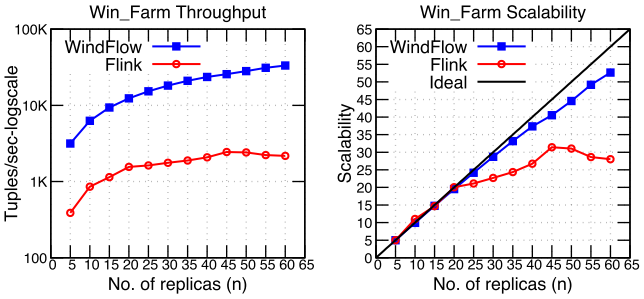



**FIGURE 15.** Throughput and scalability of the `Win_Farm` and of its emulation in Apache Flink.

of overlapping windows, whereas it does not have sense for hopping and tumbling windows or it is ineffective when the pane length is too small.

### B. FIRST CASE STUDY: SPATIAL QUERY

In the first part of the evaluation, we use an application belonging to the broad class of spatial queries. The input stream is produced by an operator in charge of receiving directly packets from a network device and doing frequency feature extraction from their payload. Six different attributes are produced, each one reporting the frequency of a specific dictionary item in the payload useful for malware detection. Such input tuples fed a windowed operator configured in order to use temporal windows of one second with a slide of 10 milliseconds.

The operator computes the *skyline* query [22]. Each input is interpreted as a 6-dimensional point, and the query returns the tuples that are not dominated by any other tuples in the same window. We say that tuple $t$ dominates tuple $t'$ when it is no worse than $t'$ in the values of all the dimensions and better in at least one dimension. We use the skyline computation implemented with the `BSkyTree` algorithm [23], available in the `SkyBench` library downloadable from GitHub.[3]

In our implementation, the call to the `BSkyTree` algorithm has been encapsulated in a lambda function doing the data-layout conversion needed to interface the `SkyBench` implementation. Furthermore, we generate the timestamps with a high input rate (of 300K tuples per second) in order to be able to study the scalability without being limited by the input pressure. The best results for this query are obtained by mapping at most one thread per physical core without using the additional hyper-threading contexts. For this reason, the number of threads will not exceed 64.

#### 1) PROTOTYPING A FIRST SOLUTION

In the application, input records are not grouped by any key attribute. Therefore, we cannot apply the `Key_Farm` pattern to increase throughput. The first solution is to perform the `SkyBench` algorithm in parallel on different windows by leveraging the `Win_Farm` pattern. We report in Figure 15 the results of a set of experiments where we tested various replication degrees (we denote this version as `WF(n)`, that is the `Win_Farm` with $n > 0$ `Win_Seq` replicas). Since two threads are used for generating the stream and for absorbing window results, while two threads are used for the emitter and collector nodes, the use of 60 replicas allows us to use all the physical cores of the machine.

The WindFlow implementation is able to process about 650 tuples/sec with parallelism one, and the peak throughput grows steadily by increasing the number of replicas reaching about 33,000 tuples/sec with `WF(60)`, see Figure 15(left). The maximum scalability (i.e. ratio between the throughput with $n \geq 1$ replicas and the one with $n = 1$) is up to $52\times$ and, as shown in Figure 15(right), it is roughly ideal and slightly deteriorates with more than 30 replicas.

We emulate the behavior of the `Win_Farm` pattern in Apache Flink, a popular SPS which is generally considered efficient also for single-machine execution [24]. The windowed operator is configured with several replicas each one using a sliding parameter that is $n$-times greater than the original one. Furthermore, a `flatmap` operator is in charge of performing the complex distribution shown in Algorithm 2 by delivering each tuple to all the replicas of the windowed operator assigned to windows containing that specific tuple. To be sure that copies of the same tuple are delivered to the right replicas, we fill a special `tag` attribute of the tuple which is used to emulate the distribution with a key-based scheduling. Furthermore, the replicas of the `flatmap` operator are chained with the ones of the windowed operator (i.e. they are executed by the same threads).

The performance of the Apache Flink emulation with one replica is one order of magnitude lower than WindFlow. Furthermore, the maximum scalability is $30\times$ with 45 replicas and stops increasing with higher parallelism because the `flatmap` operator, though parallel, becomes a bottleneck.

#### 2) EXPLOITING WINDOWS OVERLAPPING

The *transitivity* property holds [22], that is if $t_1$ dominates $t_2$ and $t_2$ dominates $t_3$, then $t_1$ dominates $t_3$. This property can be exploited to split the skyline computation into independent computations of sub-skylines on window partitions, and a

[3]Downloadable form https://github.com/sean-chester/SkyBench.

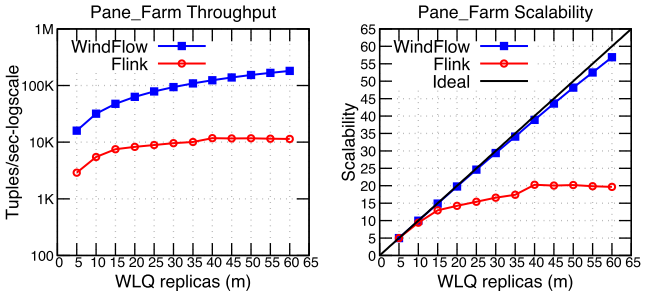



**FIGURE 16.** Throughput and scalability of the `Pane_Farm` and of its emulation in Apache Flink.

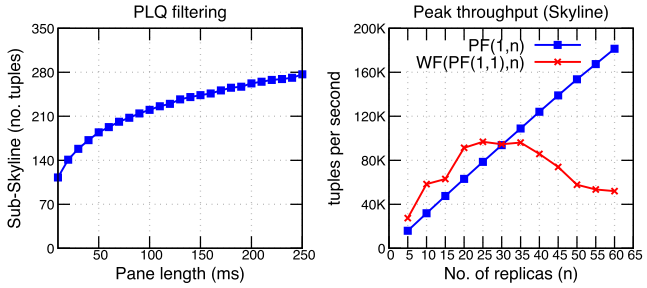



**FIGURE 17.** Average size of the sub-skyline with different pane lengths (left), and peak throughput of `WF(PF(1,1),n)` compared with `PF(1,n)` (right).

final computation produces the window-wise skyline. This allows the `Pane_Farm` pattern to be instantiated for the query, with the PLQ stage in charge of computing the sub-skylines of panes spanning 10 milliseconds of length.

The `Pane_Farm` pattern is executed with a sequential PLQ stage (sufficient to process 10-ms panes without being a bottleneck) and a parallel WLQ stage. The `Pane_Farm` is used with the transformation level `LEVEL1`, thus merging the `Win_Seq` instance in the PLQ stage with the emitter node of the WLQ. We refer to this solution as `PF(1,m)`.

This solution reduces the number of comparisons. The PLQ stage processes 10-ms panes that contain approximately 3000 tuples each. Among those inputs, only 4% on average are included in the resulting sub-skyline, thus making the PLQ an aggressive filter of dominated tuples. This alleviates the computational burden in the WLQ stage, where 100 panes per window are processed by comparing only a small fraction of the original tuples. As shown in Figure 16(left), the throughput of `PF(1,60)` is up to 5 times greater than the one achieved by `WF(60)`, see Figure 15(left). Also the scalability in Figure 16(right) is always close to the ideal.

The emulation in Apache Flink uses the previous emulation of the `Win_Farm` pattern for each stage. Due to the overhead of the JVM compared with running a native code, we need five replicas in the PLQ stage to prevent it from being a bottleneck. Figure 16(left) reports the throughput measured by varying the parallelism in the WLQ stage and Figure 16(right) the scalability. Although the performance is superior to the one achieved by the `Win_Farm` emulation, the scalability is limited by 20× due to the distribution overhead. Also in this case, our C++ implementation outperforms the Apache Flink one both in terms of absolute performance and in scalability.

### 3) LEVERAGING NESTED SOLUTIONS

To further increase throughput, the user could try to nest `Pane_Farm` instances within an outermost `Win_Farm` pattern. This idea cannot be easily implemented in Apache Flink. Therefore, we report the results for our library only.

The nesting changes the pane length. As an example, with two replicas the pane length becomes 20 ms instead of 10 ms. Figure 17(left) shows what happens by increasing the pane length in terms of the number of tuples selected in the sub-skylines of the panes. Although the sub-skyline cardinality increases, this happens quite slowly (e.g., 20-ms panes have sub-skylines only 25% larger than the ones of 10-ms panes). This reduces the comparisons done in the WLQ stage where there are fewer panes per window.

To use larger panes, we have to increase the outermost replication degree as much as possible. We replicate a `PF(1,1)` instance where both the PLQ and the WLQ stages are sequential. Furthermore, to minimize the number of threads, we enable transformation level `LEVEL1` that combines the two stages sequentially by a single FastFlow node (see Figure 12). This solution is referred as `WF(PF(1,1),n)`, where `PF(1,1)` is replicated $n > 0$ times within the `Win_Farm`. Although the structure is similar to the one of `WF(n)`, the internal sequential instances reuse results of previously calculated panes to produce window results, so they provide a more efficient window processing. This is reflected with a higher peak throughput sustained by this solution with a number of replicas less than 30, then the throughput drops quickly with more parallelism, see Figure 17(right). The reason is that the emitter of the `Win_Farm` pattern starts to be a bottleneck. Indeed, according to the $(w, s)$ query parameters, each tuple belongs to about 100 consecutive windows thus forcing the emitter to do a large broadcast of each tuple each time. We observe that this problem was hidden in the `WF(n)` and `PF(1,m)` versions. In the first case, because the inner replicas were slower and this problem did not arise up. In the second case, because the PLQ stage greatly reduced the number of tuples reaching the WLQ stage, making the input pressure to the emitter of the WLQ stage substantially lower.

To balance the advantage of using larger panes with the distribution overhead, we study the use of other complex nested solutions. If the `Pane_Farm` replicas within the `Win_Farm` are made more parallel, we would need a smaller number of replicas by reducing the distribution overhead in the emitter, although panes are now smaller. Figure 18 summarizes the peak throughput obtained by some of such nested configurations. We chose the replication degrees in order to use as many cores as possible of the machine, except for `WF(PF(1,1),n)` in which we consider the replication degree of the `Win_Farm` achieving the best throughput in Figure 17(right) (that is around 25 replicas).

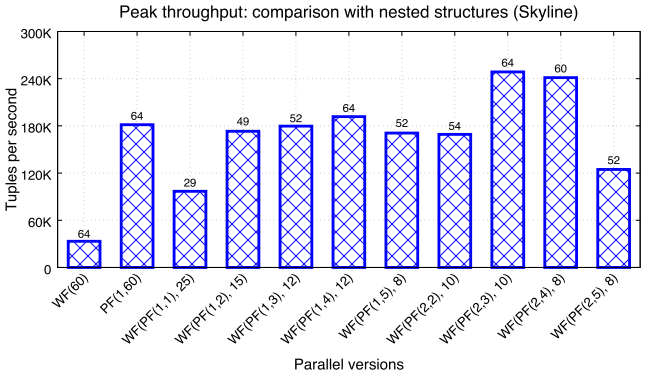



**FIGURE 18.** Peak throughput achieved by different nested structures of `Win_Farm` and `Pane_Farm` patterns.

We report above each bar the number of cores used by each configuration. Using more threads than cores (i.e. hyper-threading) is detrimental in this application. The internal `Pane_Farm` replicas are configured with transformation level `LEVEL2`, while the outermost `Win_Farm` has only one collector node (merging emitter nodes impairs performance for this application). As we can see in the Figure 18, some nested structures are not able to use all the cores of the machine although they reach a similar peak throughput than the `Pane_Farm` on all the cores. Instead, the configurations `WF(PF(2,3),10)` and `WF(PF(2,4),8)` are particularly effective; they use almost all the cores and the peak throughput is about 35% greater than using the `Pane_Farm` alone, which gives a practical justification of the effectiveness of the complex nested structures allowed by the library. We observe that without and automatic tuning support it might be difficult by the user to find a nested configuration exploiting all the cores (if it exists). In this sense, we will study in our future research how to exploit the potential of task-based parallelism in the RTS, since the decoupling between tasks and threads can help to solve this issue.

### C. SECOND CASE STUDY: YSB

We study the achieved performance on the popular *Yahoo! Streaming Benchmark* [25] (briefly, `YSB`). The original benchmark emulates a simple advertisement application, where tuples are consumed from `Kafka` (a publish-subscribe system) and results are committed into `Redis` (a key-value store). To avoid external services becoming a bottleneck limiting the overall performance, we have used the implementation adopted in prior work [4], [11] where the interactions with `Kafka` and `Redis` have been removed and data generation and processing is performed locally.

The application structure is depicted in Figure 19. The filter applies a predicate to drop all the tuples with type not equal to "`view`", while the flatmap joins each received tuple with the value of the corresponding entry in an in-memory hash table populated before starting the processing. For the WindFlow version, we implemented the operators using raw FastFlow nodes, while the sliding-window aggregate is implemented by instantiating the `Key_Farm` pattern with the incremental

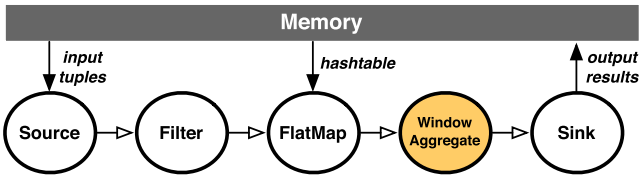



**FIGURE 19.** Yahoo! streaming benchmark.

interface. The aggregate is a count applied on tumbling windows of 10 seconds. To build windows, tuple are grouped by the identifier (key) of the advertisement campaign (100 distinct keys are present in the original code).

Figure 21 shows a comparison where the peak throughput obtained by WindFlow is compared with the one achieved using Apache Flink and Apache Storm. The experiments were run on the Intel KNL. Since hyperthreading provides some performance benefits in this application, we consider the maximum number of threads equal to the number of hyper-threading contexts (256).

Our library provides higher throughput, more than 3× higher than Apache Flink, while Apache Storm is not efficient on single servers as already proved in prior works [24]. The higher performance obtained by WindFlow derives from the RTS of our library. Our implementation is more lightweight, since it avoids data serialization into single machines and makes use of efficient lock-free queues for data forwarding (see Section III).

We have also collected the latency measurements obtained by running our patterns against Apache Flink and Apache Storm. The latency of a result is the elapsed time from when the last tuple of a window has been generated by the source, to when the aggregate of the corresponding window is received by the sink. Since the source generates at full speed, the latency is also affected by the enqueueing of tuples in the intermediate operators and in the backpressure mechanism adopted by the SPS. The results are shown in Figure 20. The latency with WindFlow is of few milliseconds while it is hundreds of milliseconds or event thousands of milliseconds in Apache Flink and Apache Storm. The variability of the latency increases with more parallelism in the RTS.

As an additional experiment, we modified the YSB in order to generate events *all belonging to the same campaign*. In this single-keyed scenario, Apache Flink and Apache Storm are both unable to natively support parallelism for the window aggregate operator, since replicas work in parallel only on distinct key groups. With our library the programmer can use other patterns to circumvent the problem. The `Win_Farm` and `Pane_Farm` patterns are ineffective because the benchmark utilizes non-overlapping windows. So, following the design space exploration in Fig. 14, we selected the `Win_MapReduce` pattern as the best fitting solution for the problem, still able to compute the window aggregates in parallel. The results are summarized in Figure 22. In the two JVM-based SPSs and in WindFlow with the `Key_Farm` pattern, the throughput stops increasing with low parallelism because the aggregate operator becomes

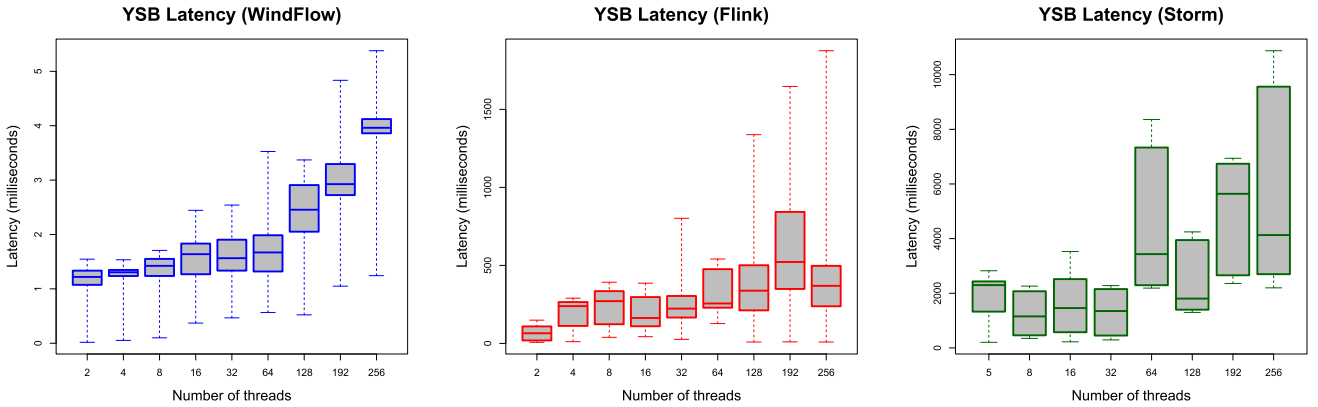



**FIGURE 20.** Measured latencies (YSB): We report in each bar the maximum and minimum measures, the 75-th, 50-th and 25-th percentile. We point out the different timescales on the y-axis.

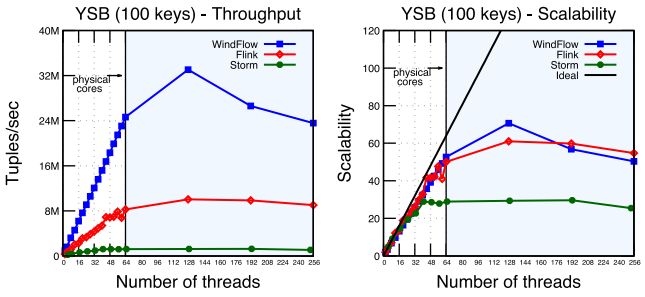



**FIGURE 21.** Throughput (YSB with 100 keys).

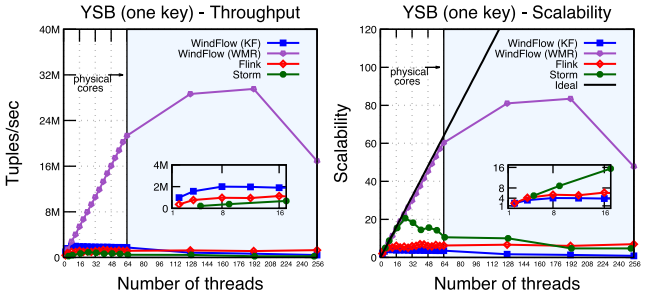



**FIGURE 22.** Throughput comparison (YSB one key).

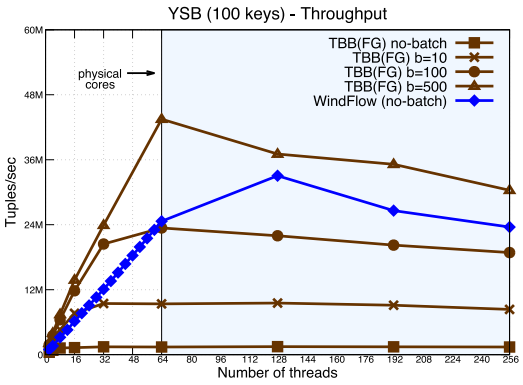



**FIGURE 23.** Throughput comparison between the FastFlow and the Intel TBB implementation of YSB.

a bottleneck. Our solution that replaces the `Key_Farm` pattern with a `Win_MapReduce` instance in our library allows the throughput to steadily increase with higher parallelism, reaching values (about 30M tuples/sec) close to the ones measured in Figure 21.

### 1) EXPERIENCE WITH OTHER RTS MODELS

In this part, we propose an experiment to assess the suitability of other models (see Section II-D) to implement the RTS. We have implemented the YSB in Intel TBB (version `2019`, update `6`) with the FlowGraph interface. The sliding-window logic has been emulated for the specific purpose of the YSB. The source node (in a parametric number of replicas) has been implemented with the `source_node` class while for the other operators we used the `function_node` and `multifunction_node` classes. For some nodes (i.e. filter and flatmap) we chose an `unlimited` concurrency limit (i.e. number of tasks spawnable per node), while for the aggregate we used one node for each existing key (100).

Figure 23 reports the throughput with the Intel TBB implementation (`TBB(FG) no-batch`) against our original implementation (`WindFlow no-batch`). We vary the number of threads (for Intel TBB this is the size of the thread pool). Furthermore, we enabled the use of the Intel TBB allocator by properly setting the `LD_PRELOAD` environmental variable. As we can observe, the throughput of the Intel TBB version is very low (up to 3M tuples/second). The reason is that all the nodes in the YSB are very fine-grained (less than one microsecond) and the overhead of task scheduling prevents to achieve good performance (this is also obtained by enabling the `lightweight` policy to avoid spawning a new task for each message).

Two options can be followed to increase the task grain. The first is to merge different nodes, which is not an effective alternative for the YSB since there are few operators, and all of them are very fine-grained. The second is to leverage the *micro-batching* technique: i.e. each task, instead of processing one input at a time, executes the processing on $b > 0$ successive inputs and forwards the corresponding

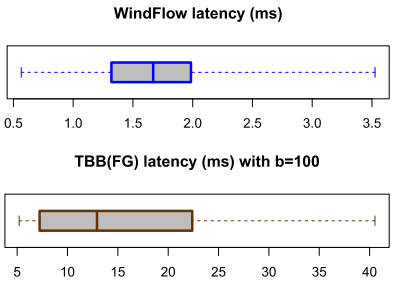



**FIGURE 24.** Latency comparison between the FastFlow and the Intel TBB implementation of YSB.

outputs to the next node as a single message. By tuning the parameter $b$, we can increase the grain size. As shown in Figure 23, a small batch of few hundreds of inputs is sufficient to achieve ideal scalability by the `TBB(FG)` implementation and higher throughput measurements (more than 40M tuples/sec). Of course, even if the batch size is small, it has some negative implications on the latency, which increases with greater values of $b$ because each input cannot be immediately processed upon its generation (we must wait the filling of the batch). Figure 24 shows the latency between `TBB(FG) b=100` and `WindFlow no-batch` with 64 threads, two configurations having similar throughput (however, the `TBB(FG)` latency is higher for all tested values of $b$). Because of the batching, the latency is one order of magnitude higher than the one with `WindFlow no-batch`, and this suggests that the task-grain size should be adjusted dynamically to provide a good tradeoff between throughput and latency so to maintain the latter as small as possible. This can be a promising direction to follow to incorporate a task-based RTS as an alternative backend, and to enable a comprehensive study of its potential in terms of load balancing that we will target in our future research.

Finally, we remark that the micro-batching technique could also be applied in the FastFlow implementation to further increase its throughput even if at the price of higher latency. Since, this feature is not yet transparently offered to the WindFlow user (the user should implement the micro batching within the business logic of the nodes), we decided not to study this case leaving this analysis as future research.

## VII. RELATED WORK

SPSs traditionally target distributed systems of multicore-based nodes. Recently, this vision has been criticized because existing SPSs do not efficiently exploit modern scale-up servers equipped with tens of cores and terabytes of memory. The limits of existing systems have been clearly highlighted in [24], where the behavior of SPSs has been precisely analyzed in terms of cache misses, NUMA effects, and the intrinsic overhead of the JVM and of its Garbage Collector.

In [11] the authors have investigated how SPSs can be configured in order to exploit at best the features of modern machines. They have derived a similar conclusion that SPSs are not capable of fully exploiting the current affordable hardware and high-speed networks. One of the reasons is in the not efficient data forwarding mechanism implemented by the JVM-based SPSs, which are based on concurrent queues with conventional conditional variables that are far from being efficient in case of a high stream pressure.

SPSs optimized for scale-up scenarios have been proposed in the recent years. `Saber` [3] is based on a hybrid execution model supporting CPU+GPU. `Streambox` [4] is written in C++ and have a peculiar utilization of the memory based on NUMA regions. Both the solutions are still prototypes and have a yet incomplete interface providing support to a set of built-in operators (e.g., the ones from traditional streaming algebra) and not supporting a completely general business logic code like in Apache Flink and Apache Storm.

Orthogonal to the aforementioned issues, no serious advancement in programming abstractions has been developed in the last years for SPSs. For stateful operators the only native solution is to rely on a partitionable state and a hash distribution of input items to replicas working on different key groups. Attempts to produce more powerful abstractions, ready-to-use by the programmer, have been made in few past works. In [26] a DSL for streaming applications has been developed using the C++11 attributes to annotate the code by introducing parallel patterns (pipelines and functional replication). `GrPPi` [27] is a high-level C++11 interface for writing parallel programs featuring a pattern-based approach similar to the one proposed in this paper. However, the patterns are general and not tailored for the specific needs of streaming analytics. `PiCo` [28] is aimed at providing batch and stream processing with a unified C++ interface. It also embraces the pattern-based methodology, although patterns do not support complex nested structures.

Further attempts to provide parallel processing support have been made in topics closely related to DSP. A relevant example is the work done in the Complex Event Processing field (CEP), where input streams must be efficiently processed to detect complex events in real time. As stated in [29], CEP systems require parallel methods that are essentially different than the ones proposed in the DSP domain. For this reason, the authors of [29] propose a parallel processing approach that splits the input stream into substreams that can be processed in parallel (as in the `Key_Farm` pattern in the present paper). However, such splitting is not simply based on the existence of a key attribute, but it is based on advanced heuristics able to balance the workload by still providing accurate detection of complex events in parallel. Extending the idea of our composable and nestable patterns to the CEP domain, starting from this existing experience, is a further promising direction to follow.

## VIII. CONCLUSION AND FUTURE WORK

Parallelism abstractions provided as parallel patterns are executable components that the user can instantiate with the business logic code and that have a clear parallel semantics. This paper tried to bring this idea into the DSP domain. Parallel patterns for sliding-window streaming analytics have

been provided through a C++ fluent interface, and can be built in a modular way to express nested structures modeling a complex parallelism exploitation pattern. Our approach has been prototyped with the WindFlow library based on FastFlow and its thread-based parallelism model, showing significant performance improvements compared with the emulation of the patterns in existing JVM-based SPSs.

The last part of the paper provided a preliminary study which is influential for our future research. The implementation of the YSB in Intel TBB showed that the task-based parallelism, popular in the High-Performance Computing community, can be profitably used to obtain good performance in streaming analytics applications. However, this requires careful and possibly dynamic use of micro-batching techniques to reach the required task granularity to scale well on shared-memory machines without blowing up the latency. Investigating this issue, and how to make WindFlow a multi backend library, is a priority in our future work.

## REFERENCES


[1] A. Jain, *Mastering Apache Storm: Real-Time Big Data Streaming Using Kafka, Hbase and Redis*. Birmingham, U.K.: Packt, 2017.

[2] E. Friedman and K. Tzoumas, *Introduction to Apache Flink: Stream Processing for Real Time and Beyond*, 1st ed. Newton, MA, USA: O'Reilly Media, Inc., 2016.

[3] A. Koliousis, M. Weidlich, R. C. Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "SABER: Window-based hybrid stream processing for heterogeneous architectures," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2016, pp. 555–569. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882906

[4] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "StreamBOX-HBM: Stream analytics on high bandwidth hybrid memory," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, New York, NY, USA, 2019, pp. 167–181. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304031

[5] T. De Matteis and G. Mencagli, "Parallel patterns for window-based stateful operators on data streams: An algorithmic Skeleton approach," *Int. J. Parallel Program.*, vol. 45, no. 2, pp. 382–401, Apr. 2017. doi: 10.1007/s10766-016-0413-x.

[6] G. Mencagli, M. Torquati, F. Lucattini, S. Cuomo, and M. Aldinucci, "Harnessing sliding-window execution semantics for parallel stream processing," *J. Parallel Distrib. Comput.*, vol. 116, pp. 74–88, Jun. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517302976

[7] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: High-level and efficient streaming on multicore," in *Programming Multicore and Many-Core Computing Systems*. Hoboken, NJ, USA: Wiley, 2017, pp. 261–280. [Online]. Available: http://dx.doi.org/10.1002/9781119332015.ch13

[8] T. G. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, 1st ed. Reading, MA, USA: Addison-Wesley, 2004.

[9] B. Gedik, "Generic windowing support for extensible stream processing systems," *Softw., Pract. Exper.*, vol. 44, no. 9, pp. 1105–1128, Sep. 2014. doi: 10.1002/spe.2194.

[10] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*, 1st ed. New York, NY, USA: Cambridge Univ. Press, 2014.

[11] S. Zeuch, B. Del Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, "Analyzing efficient stream processing on modern hardware," *Proc. VLDB Endowment*, vol. 12, no. 5, pp. 516–530, Jan. 2019. doi: 10.14778/3303753.3303758.

[12] Z. Nabi, *Pro Spark Streaming: The Zen of Real-Time Analytics Using Apache Spark*, 1st ed. Berkely, CA, USA: Apress, 2016.

[13] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2012.

[14] M. Voss, R. Asenjo, and J. Reinders, *C++ Parallel Programming With Threading Building Blocks*, 1st ed. New York, NY, USA: Apress, 2019.

[15] C. Campbell and A. Miller, *A Parallel Programming With Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*, 1st ed. Redmond, WA, USA: Microsoft Press, 2011.

[16] D. Wyatt, *Akka Concurrency*. Walnut Creek, CA, USA: Artima Incorporation, 2013.

[17] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting actor programming in C++," *Comput. Lang. Syst. Struct.*, vol. 45, pp. 105–131, Apr. 2016. doi: 10.1016/j.cl.2016.01.002.

[18] M. Torquati, "Harnessing parallelism in multi/many-cores with streams and parallel patterns," Ph.D. dissertation, Univ. Pisa, Pisa, Italy, 2019.

[19] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Proc. 18th Eur. Conf. Parallel Process.*, in Lecture Notes in Computer Science, vol. 7484. Berlin, Germany: Springer, Aug. 2012, pp. 662–673.

[20] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams," *ACM SIGMOD Rec.*, vol. 34, no. 1, pp. 39–44, Mar. 2005. doi: 10.1145/1058150.1058158.

[21] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, Apr. 2014, Art. no. 46. doi: 10.1145/2528412.

[22] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proc. 17th Int. Conf. Data Eng.*, Washington, DC, USA, Apr. 2001, pp. 421–430. [Online]. Available: http://dl.acm.org/citation.cfm?id=645484.656550

[23] J. Lee and S.-W. Hwang, "Scalable skyline computation using a balanced pivot selection technique," *Inf. Syst.*, vol. 39, pp. 1–21, Jan. 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437913000744

[24] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze, "Revisiting the design of data stream processing systems on multi-core processors," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 659–670.

[25] DataArtisans. (2016). *Extending the Yahoo! Streaming Benchmark*. [Online]. Available: https://github.com/dataArtisans/yahoo-streaming-benchmark

[26] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: A DSL for high-level and productive stream parallelism," *Parallel Process. Lett.*, vol. 27, no. 1, 2017, Art. no. 1740005. doi: 10.1142/S0129626417400059.

[27] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García, "A generic parallel pattern interface for stream and data processing," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 24, 2017, Art. no. e4175. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4175

[28] C. Misale, M. Drocco, G. Tremblay, A. R. Martinelli, and M. Aldinucci, "PiCo: High-performance data analytics pipelines in modern C++," *Future Gener. Comput. Syst.*, vol. 87, pp. 392–403, Oct. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X1732681X

[29] F. Xiao and M. Aritsugi, "An adaptive parallel processing strategy for complex event processing systems over data streams in wireless sensor networks," *Sensors*, vol. 18, no. 11, p. 3732, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/11/3732