Efficient Big Data Streaming on Modern Scale-Up Servers

Dr. **Gabriele Mencagli**, PhD
Department of Computer Science, University of Pisa, Italy


MOTIVATION

A World of Streaming Data!

Batch Processing

Permanent

Static

Bounded size (though huge)

Immediately available for processing

Stream Processing

In-motion

Dynamic

Unbounded size (though huge)

Not immediately available, arrival rate

Performance!

Processing fast data flows require fast processing techniques (suitable hardware/software solutions)

Parallel Systems

Multi-Core Server

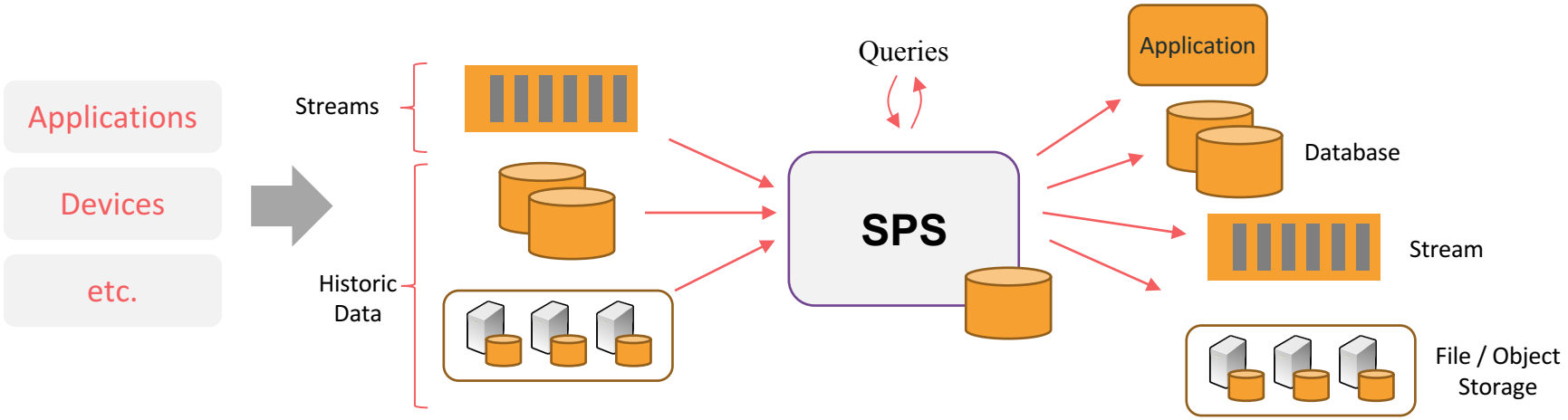
Many-Core Server

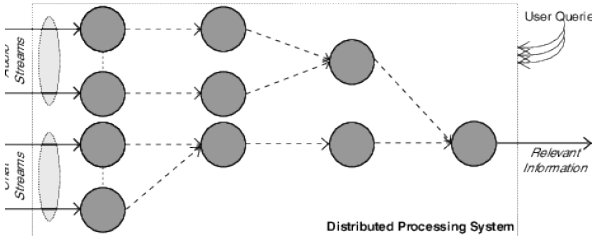
Hardware Accelerators (GPU/FPGA)

Cloud Datacenter

Stream Processing Systems

- Stream Processing Systems (**SPSs**) are powerful frameworks to develop and run streaming applications by leveraging distributed architectures (**scale-out**)
- Many open-source solutions available under the Apache umbrella





Queries are **data-flow graphs** of **operators** performing data transformations on streams

Some **Apache SPSs**

Flink

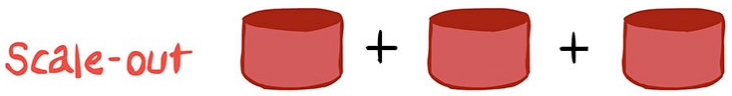
samza

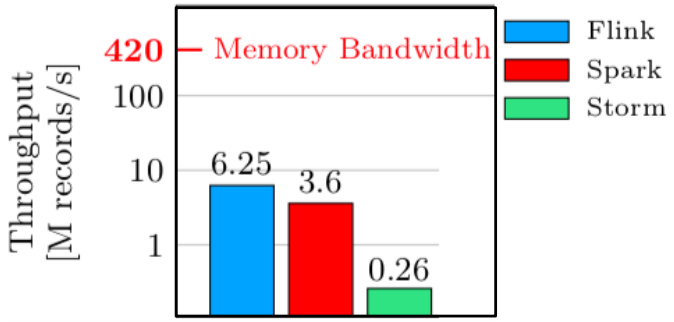
STORM

Apache HERON

Research Challenges

- **Scale-out scenarios**: traditional SPSs aim at scaling in **distributed systems**, typically clusters of homogeneous servers (multicore-based)





S. Zeuch at al. Analyzing Efficient Stream Processing on Modern Hardware. **VLDB 2019**

- They are based on the Java Virtual Machine (**JVM**)
  - high level abstraction from the underlying hardware
  - extra overhead for data access due to data (de-)serialization, virtual functions, garbage collection

Achieved performance (throughput and latency) is far from the physical limit of the single machine

- **Scale-up scenarios**: servers are equipped with **terabytes of memory**, **hundreds of cores**, **co-processors** (GPUs and FPGAs) → important alternative to process high-volume data streams with **high throughput** and **low latency** without the **overhead of distributed processing**

Solution?

Having the Right Tools

To achieve good performance…
…use the right tools!

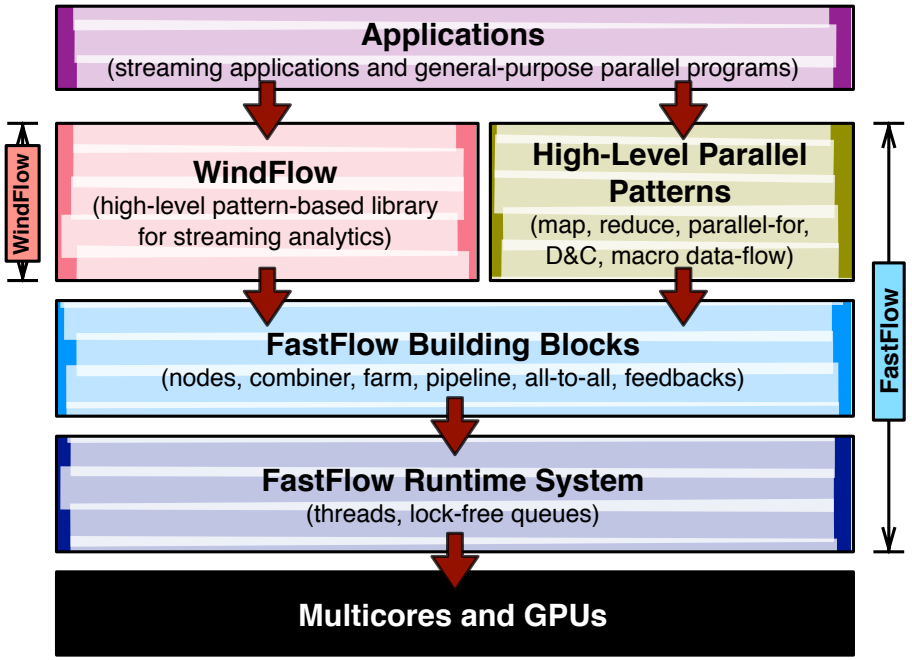
GitHub Repository
https://github.com/ParaGroup/WindFlow

Web Page

https://paragroup.github.io/WindFlow/

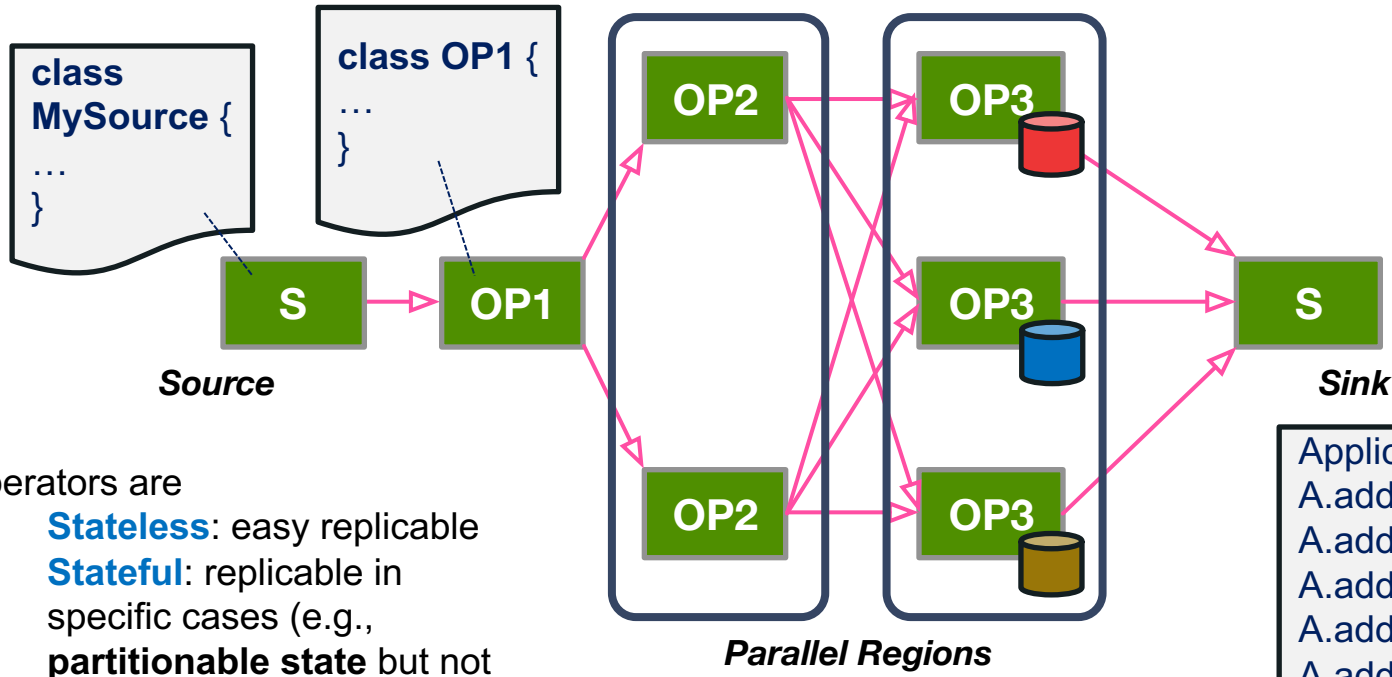
Software Layers

- WindFlow is a C++17 library for Data Stream Processing applications. The library targets scale-up servers equipped with **multi-core CPUs** (also **GPUs** but not in this talk, the work is still on-going)
- The run-time system of the library is written using the building blocks of FastFlow. Th API is user-friendly and allows the programmer to
  - Create data transformation phases
  - Interconnect them into data-flow graphs
- Software layers
  - Applications
  - **WindFlow**
  - Building blocks (**FastFlow**)
  - Raw concurrency mechanisms (**FastFlow**)
  - Multicores and GPUs
- Goals
  - Exploit at best the hardware resources
  - Avoid extra overhead
  - Low latency (**few milliseconds**, not more)



WindFlow Applications

- As in other SPSs, WindFlow allows the programmer to build an application as a data-flow graph of operators
- Operators → **vertices** doing data transformations defined by the user
- Streams → **edges** where data items are forwarded. Connections decided by the user but also partially automatic (intra-operator parallelism)



Operators are
- **Stateless**: easy replicable
- **Stateful**: replicable in specific cases (e.g., **partitionable state** but not only this case)

```
Application A = …;
A.add_source(source);
A.add(OP1);
A.add(OP2, 2);
A.add(OP3, 3);
A.add_sink(sink);
```

Operators

Source	
S → T	**Source** operator is in charge of generating a sequence of streaming items all having the same data type

Basic Operators	
T1 → M → T2	**Map** operator applies a one-to-one transformation producing one output per input
T → F → T	**Filter** operator applies a boolean predicate on each input item and drops the ones returning false
T1 → FM → T2*	**FlatMap** operator producing one or more output items per input item consumed
T1 → A → T2	**Accumulator** operator executes a "rolling" reduce or fold function on the inputs partitioned by key

Window-based Operators	
T1* → KF → T2*	**Keyed Farm** in charge of executing a windowed query in parallel on different key groups. Actually, this pattern is not limited to sliding-window computations
T1* → WF → T2*	**Windowed Farm** in charge of executing a windowed query in parallel on distinct streaming windows (key grouping is not required for parallelism)
T1* → PF → T2*	**Paned Farm** in charge of executing window-based operators in parallel by exploiting window overlapping (key grouping is not required for parallelism)
T1* → WMR → T2*	**Windowed Map-Reduce** in charge of executing window-based queries in parallel by exploiting data parallelism within each window (key grouping not required for parallelism)

Sink	
T → S	**Sink** operator is in charge of absorbing the input stream of items

C++17 Fluent Interface

- Applications are built in two steps
  - Create the operators by providing their **functional logic** and proper **configuration parameters**
  - Create the application topology through two special constructs (**MultiPipe** and **PipeGraph**)
- Operators are created
  - Using a builder object (Fluent Builder pattern): it separates the **configuration** of an operator from its **representation**
  - Builders have a fluent interface: it is possible to apply multiple properties by connecting them with ***dots*** (similarly to an **ordinary written prose**)
- **Example**

```
Operator<T1, T2> name = Operator_Builder<decltype(F)>(F)
                                                .withParam1(value)
                                                .withParam2(value)
                                                .build();
```

```
Operator name = Operator_Builder(F)
                                .withParam1(...)
                                .withParam2(...)
                                .build();
```

From **C++17**, it is possible to deduce template arguments from constructors of template classes (**Class Template Argument Deduction**)

- An instance of the operator is created with a functional logic given by F (e.g., a lambda, plain function or a functor) and with additional configuration parameters

Example of Operator Creation

- Creation of the functional logic (through a **function/lambda/functor**) and then the creation of the operator object through its Builder class

Source

```
class Source_Functor
{
public:
    ...
    // operator()
    void operator()(Shipper<tuple_t> &shipper)
    {
        for (size_t i=0; i<len; i++) {
            tuple_t t(...);
            shipper.push(t);
        }
    }
};
```

```
Source_Functor sourceF;
Source source = Source_Builder(sourceF).withName("mysource")
                                       .withParallelism(2)
                                       .build();
```

Map

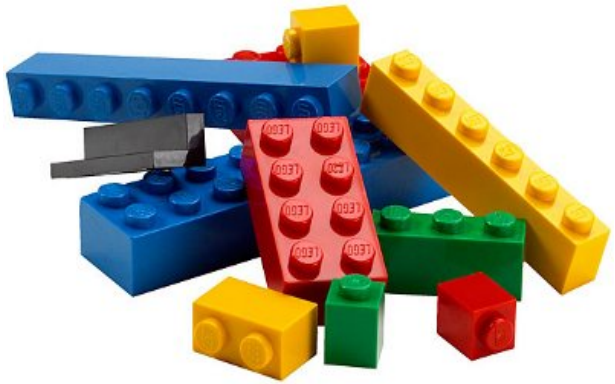
```
class Map_Functor
{
public:
    ...
    void operator()(const tuple_t &in, output_t &out) {
        // code producing the output value
        ...
        out = ...;
    }
};
```

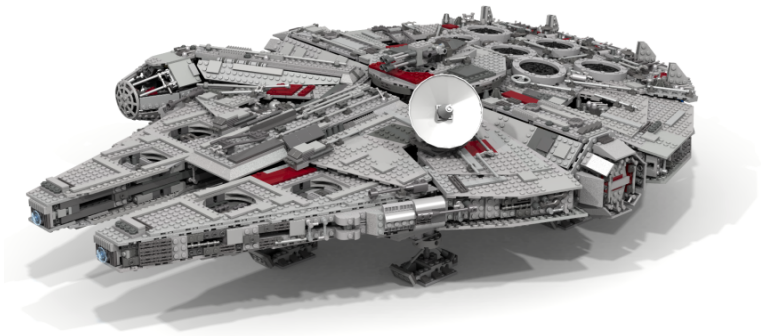
```
Map_Functor mapF;
Map map = Map_Builder(mapF).withName("test1_map")
                           .withParallelism(3)
                           .build();
```

Run-Time System

Design Idea

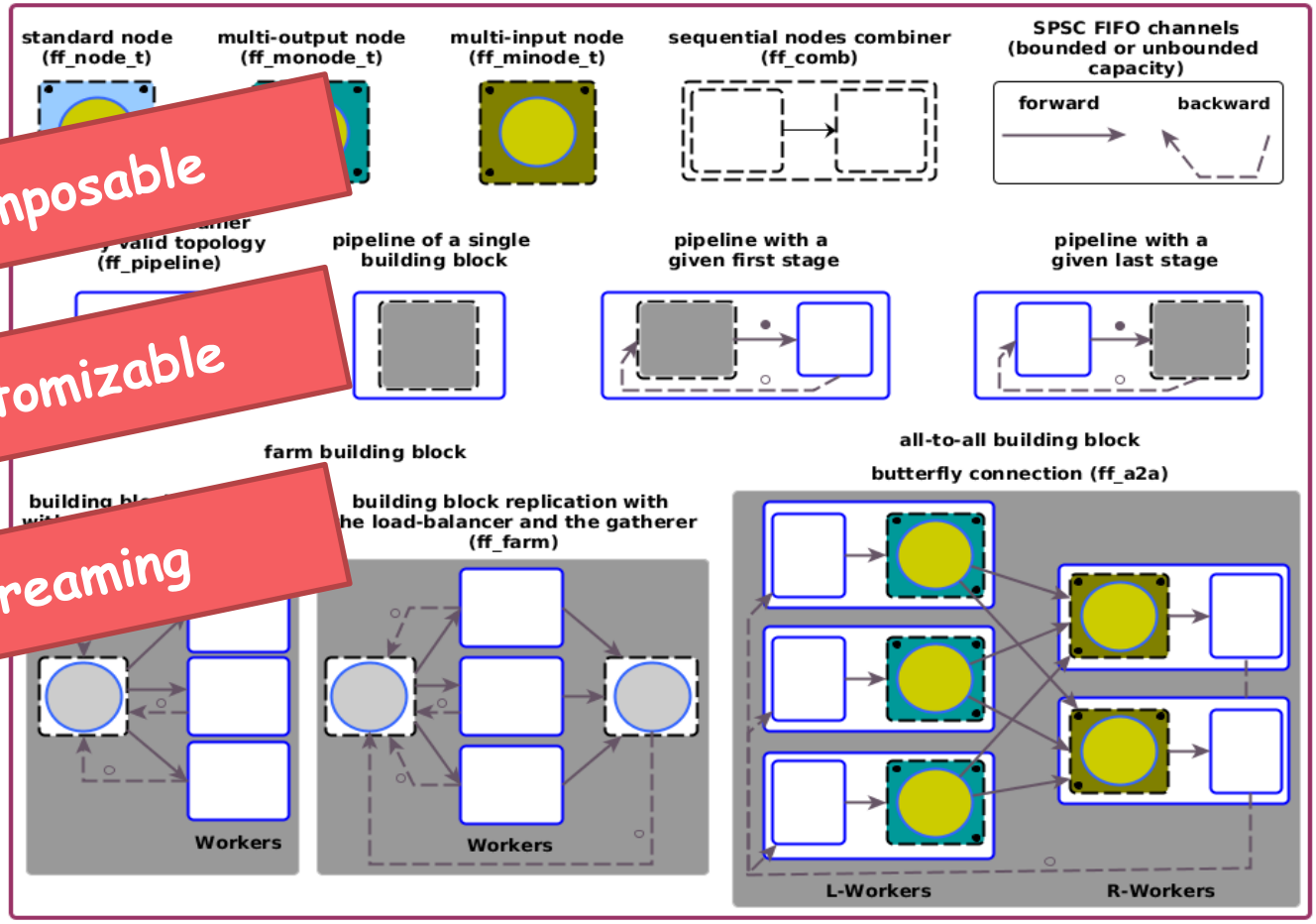
Modular design based on powerful and composable abstrations...





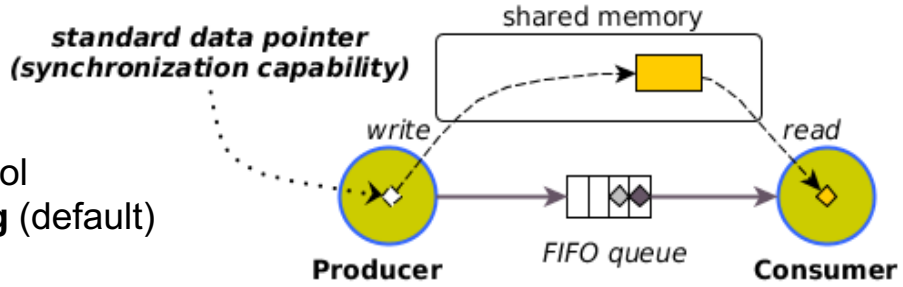
FastFlow Building Blocks (BBs)

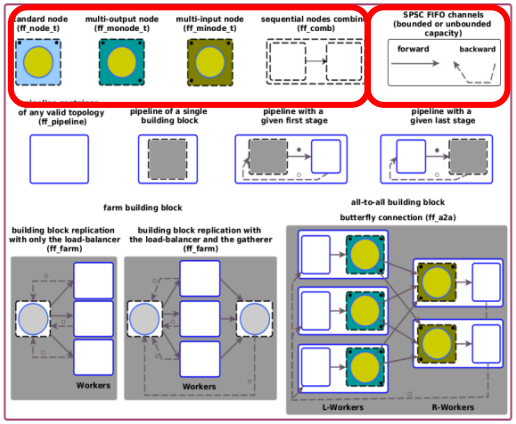
FastFlow v3.0 (GitHub https://github.com/fastflow)
Building blocks as C++ classes



Sequential BBs

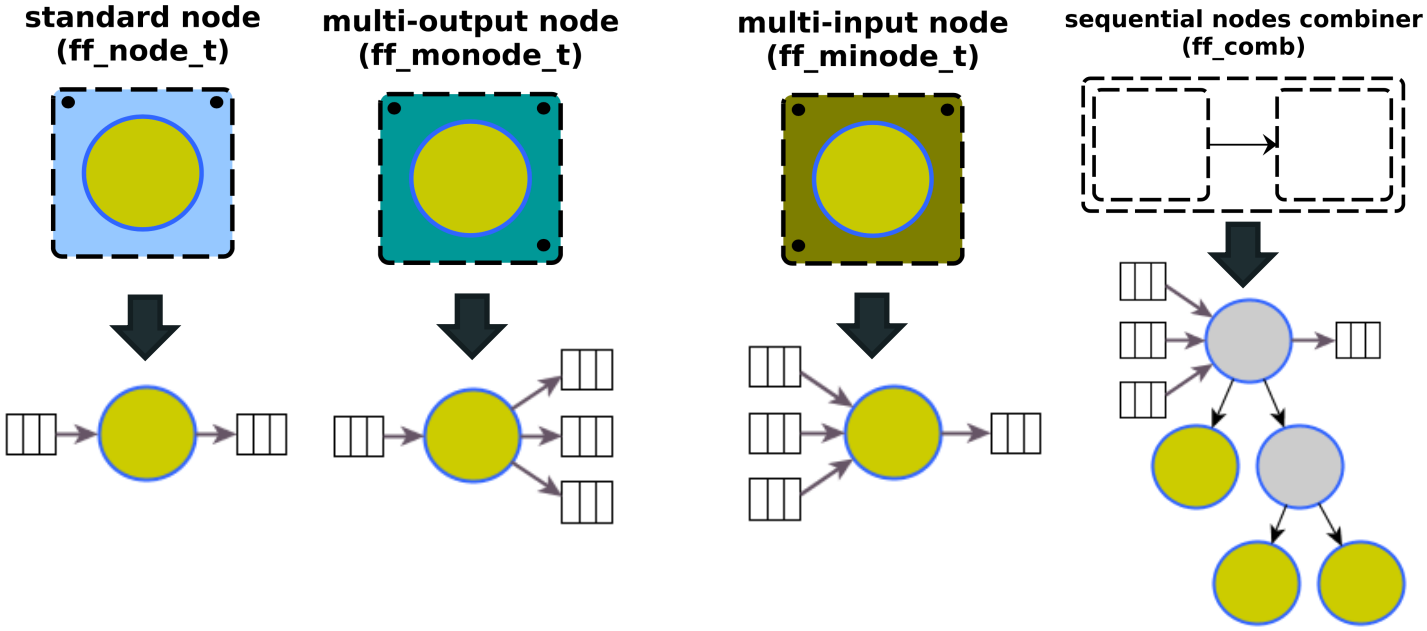
Elementary "bricks" to compose complex structures…





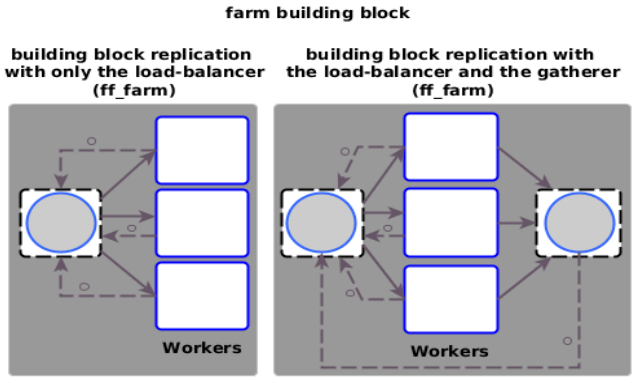
Concurrency control

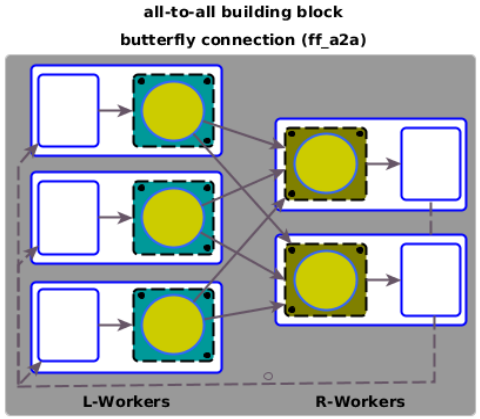
- **Non-blocking** (default)
- **Blocking**
- **Automatic/Adaptive**

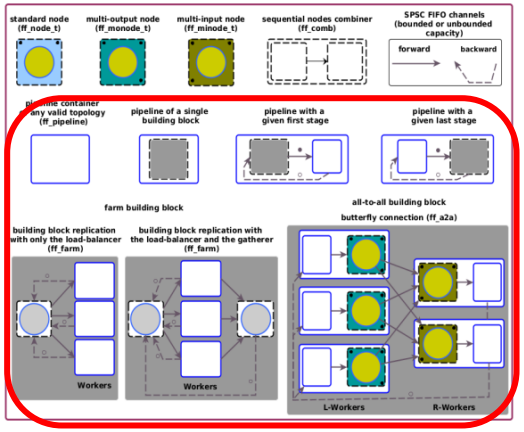


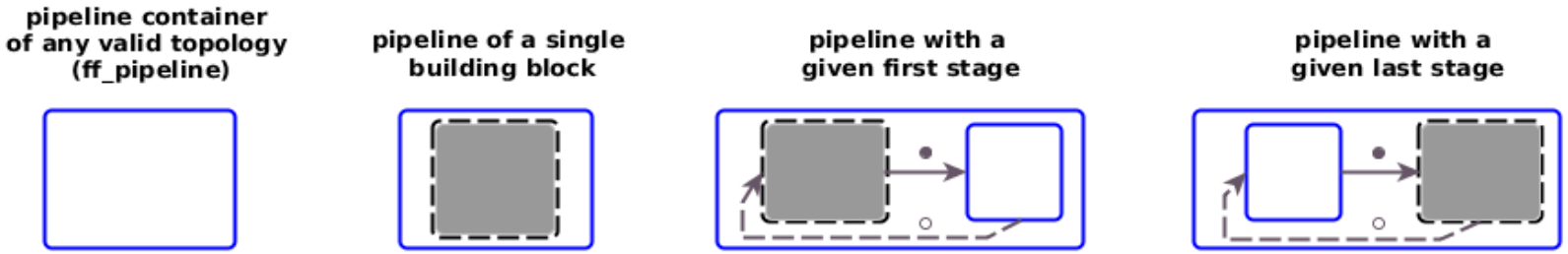
Parallel BBs

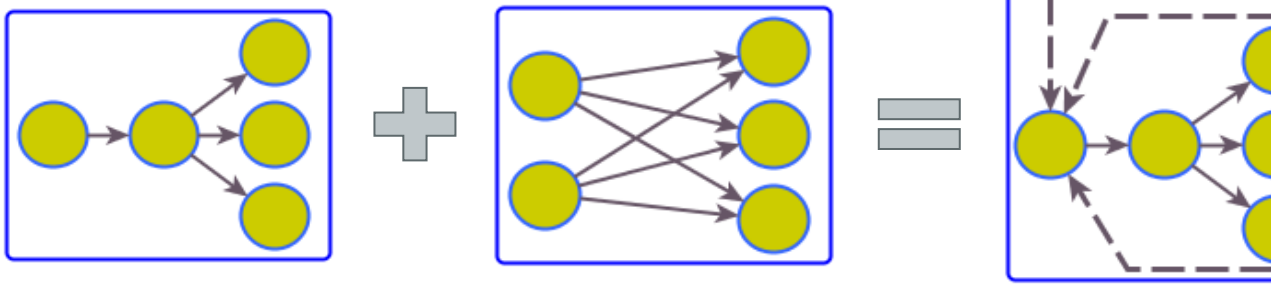
Complex "bricks" that can be composed (nested) together…





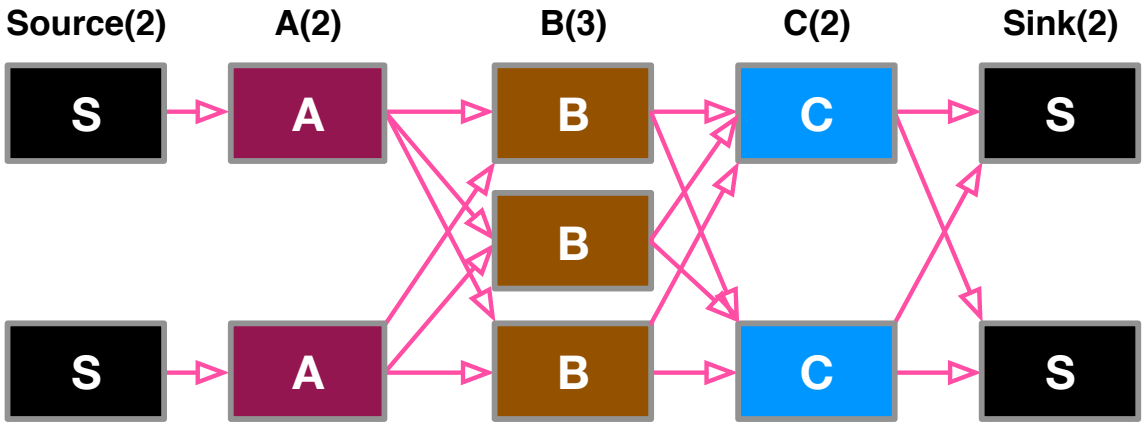






MultiPipe

- WindFlow provides a specific construct to compose together operators and to build complex streaming applications
- This construct is called MultiPipe. It is composed by
  - Some parallel pipelines working in parallel (each pipeline is a **linear sequence** of replicas of the utilized operators)
  - Pipelines might be not independent but data items may **cross** a pipeline and **jump** to another one



Two connection modes transparent to the **high-level programmer**

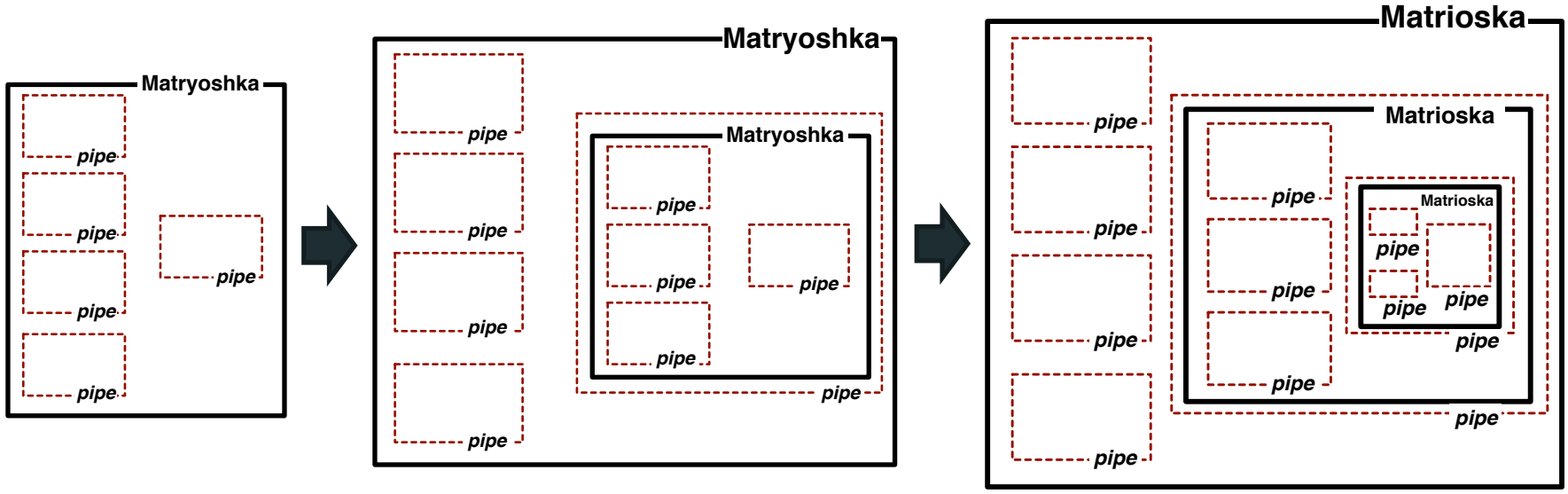
- **Direct**: one replica connected to ***exactly one*** replica of the next operator in the MultiPipe
- **Shuffle**: one replica connected to ***all*** the replicas of the next operator in the MultiPipe

- The connection mode depends on the **type** of operator and on its **parallelism level**
- Shuffle connections arise when specific distributions are employed: shuffle, key-based or complex ones depending on the next operator in the MultiPipe

Matryoshka

Matryoshka Block

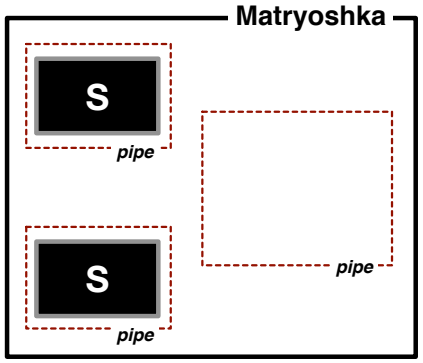
- To build a MultiPipe structure, the WindFlow run-time system adopts a **compound building block** developed using the FastFlow basic blocks → **Matryoshka**
- Such building block is **invisible to the programmer** (it is used to implement the MultiPipe). To add a new operator to the MultiPipe, a new Matryoshka can be instantiated and nested within the last Matryoshka of the MultiPipe
- **Definition**: a Matryoshka is an instance of the all-to-all block with $N > 0$ pipeline nodes in the **LEFT set** and **one** pipeline node in the **RIGHT set**



Matryoshkas nesting in the RIGHT set...

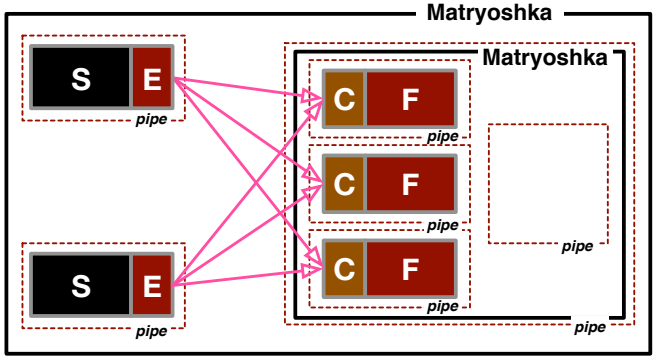
Adding Operators

- **Case 1**: a Source with parallelism $N > 0$ is added to a MultiPipe named "test1"




```
PipeGraph app("test1");
Source_Functor sourceF;
Source source = Source_Builder(sourceF).withName("test1_source")
                                       .withParallelism(2)
                                       .build();
MultiPipe &pipe = app.add_source(source);
```

- **Case 2**: applied in different cases: *i)* **different number of replicas**, *ii)* **basic operators** (Map, FlatMap, Filter, Sink) with **key-by**, *iii)* **complex operators**

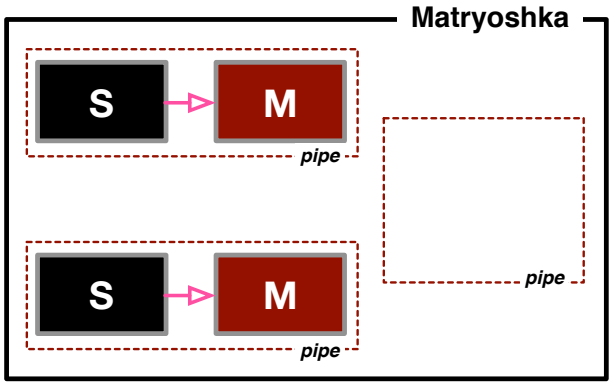



```
Filter_Functor filterF;
Filter filter = Filter_Builder(filterF).withName("test1_filter")
                                       .withParallelism(3)
                                       .build();
pipe.add(filter);
```

Shuffle connections → new Matryoshka on the RIGHT side!

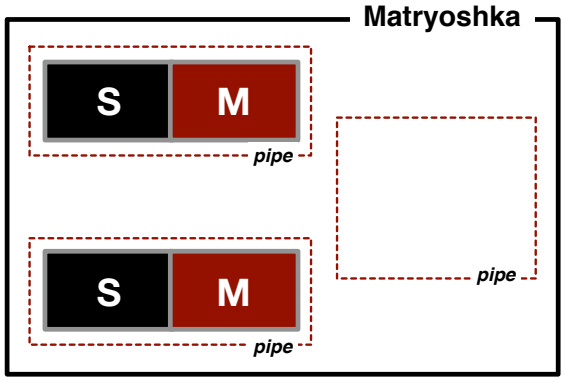
Adding/Chaining Operators

- **Case 3**: no new Matryoshka is created for some operators (**Map**, **Filter**, **FlatMap** and **Sink**) if their replicas can be appended in the pipelines of the **LEFT set**




```
Map_Functor mapF;
Map map = Map_Builder(mapF).withName("test1_map")
                           .withParallelism(2)
                           .build();
pipe.add(map);
```

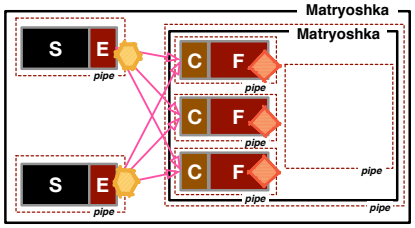
- **Case 4**: chaining avoids adding new FastFlow nodes (so additional **threads**) by replacing pointer forwarding with a more efficient function call



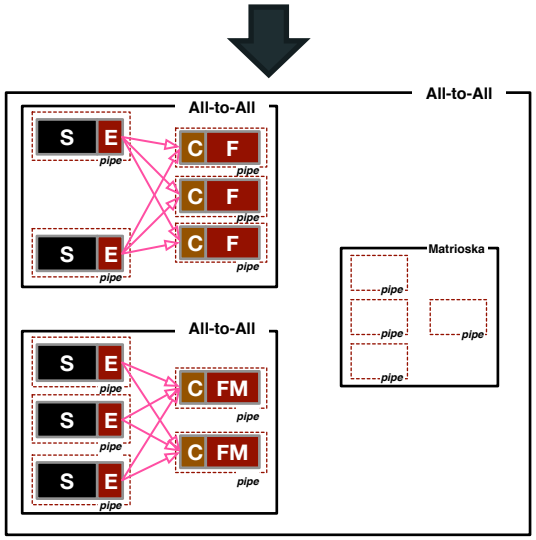

```
Map_Functor mapF;
Map map = Map_Builder(mapF).withName("test1_map")
                           .withParallelism(2)
                           .build();
pipe.chain(map);
```

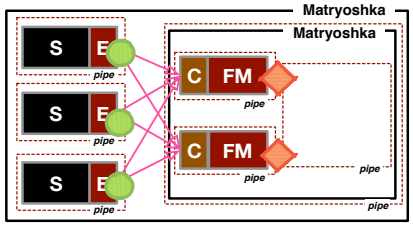
Merge of MultiPipes

- Two or more MultiPipes can be merged in order to unify the two output streams and applying common transformations on such output flow

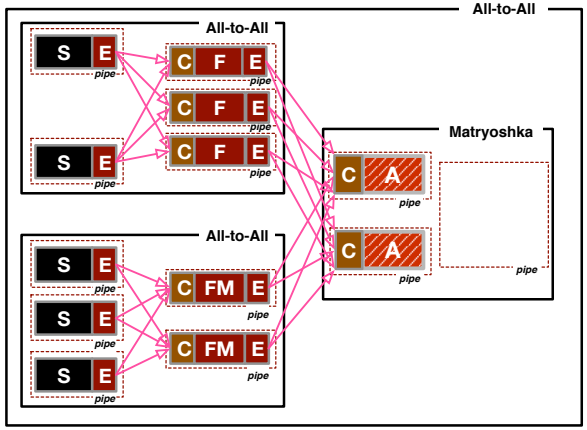



```
PipeGraph app("test_merge");
Source source1 = Source_Builder(sourceF1).withName("source1")
                         .withParallelism(2)
                         .build();
MultiPipe &pipe1 = app.add_source(source1);
Filter filter = Filter_Builder(filterF).withParallelism(3)
                           .withName("filter")
                           .build();
pipe1.add(filter);
```





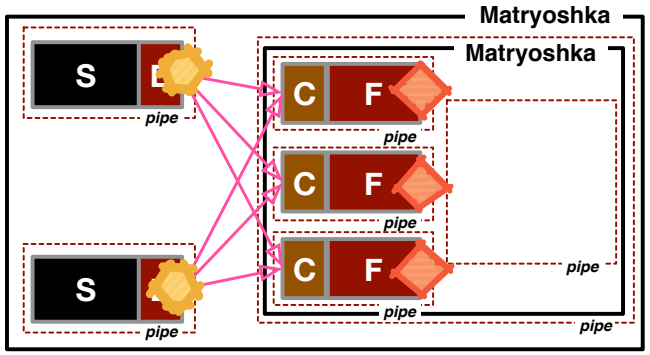

```
Source source2 = Source_Builder(sourceF2).withName("source2")
                         .withParallelism(3)
                         .build();
MultiPipe &pipe2 = app.add_source(source2);
FlatMap flatmap = Filter_Builder(flatmapF).withParallelism(2)
                           .withName("flatmap")
                           .build();
pipe2.add(filter);
```



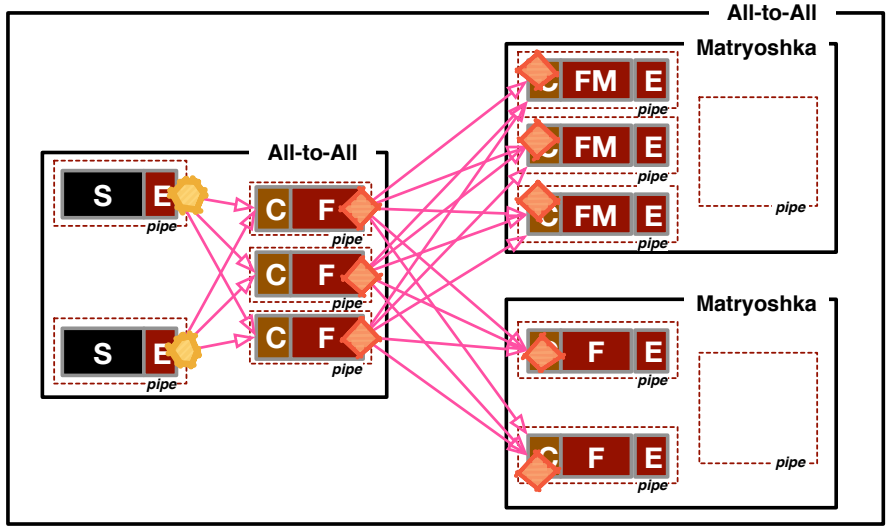

```
MultiPipe &pipe3 = pipe1.merge(pipe2); // merge of pipe1 with pipe2
Accumulator accumulator = Accumulator_Builder(foldF).withParallelism(2)
                                .withName("accumulator")
                                .build();
pipe3.add(accumulator);
```

Split of MultiPipes

- A MultiPipe can be split into several branches that are in turn other MultiPipe instances where the programmer can add further operators
- The user provides a splitting function



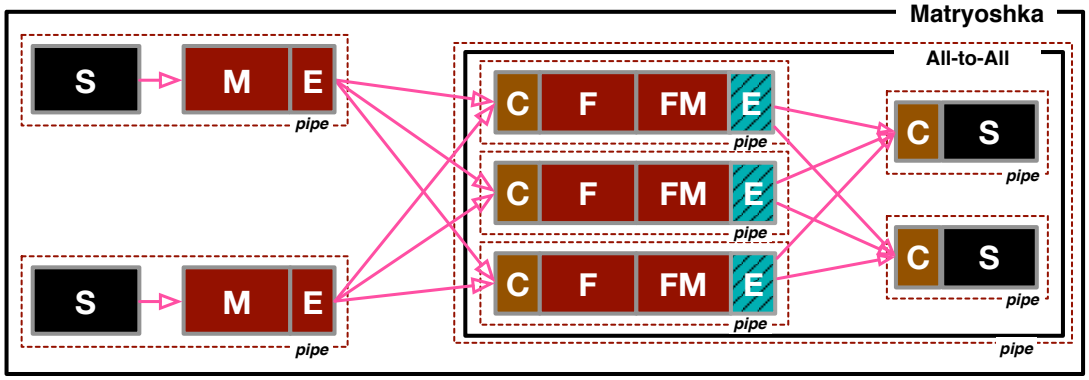

```
PipeGraph app("test_split");
MultiPipe &pipe1 = app.add_source(source);
pipe1.split([](const tuple_t &t) {
                if (t.key == "key1")
                    return 0;
                else
                    return 1;
            });
MultiPipe &pipe2 = pipe1.select(0);
pipe2.add(flatmap);
MultiPipe &pipe3 = pipe1.select(1);
pipe3.add(filter);
```



Compatible Types!

Running the Application

- Once the application has been developed, the programmer can run it through the PipeGraph (a sort of **streaming environment** like in Flink and Storm)
- The whole graph of MultiPipes must be runnable
  - Each MultiPipe should be connected in **input** with other MultiPipes, or it must have a Source
  - Each MultiPipe should be connected in **output** with other MultiPipes, or it must have a Sink
- Example



```
PipeGraph app("test1");

... // add/chain of other operators

Sink sink = Sink_Builder(sinkF).withParallelism(2)
                               .withName("test1_sink")
                               .build();
pipe.chain(sink);
app.run();
```

How many FastFlow nodes?

- **24** raw nodes (excluded combiner nodes)

How many threads?

- **9** threads are used and **pinned** on the same amount of **cores** of the multi-core CPUs

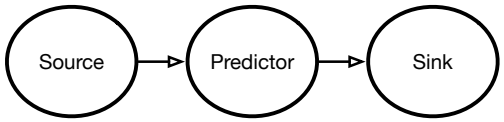
Evaluation

WINDFLOW

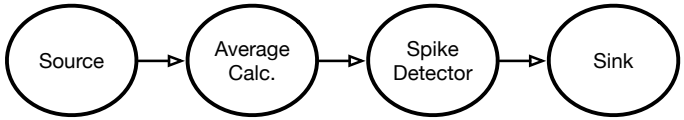
Applications

- We study **four applications** widely recognized by the Data Stream Processing community and used for the evaluation in several existing papers in the past
- Each application has been implemented using some WindFlow operators connected with a **MultiPipe**

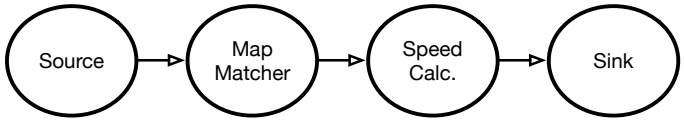
FraudDetection (FD): detects credit card transactions that might be frauds with high probability. Predictor implements a stochastic model (Markov Chain)



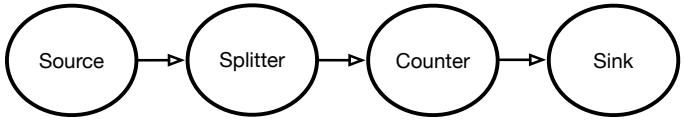
SpikeDetection (SD): monitoring of sensor data and real-time detection of measurements that deviate from the recent mean more than a specific threshold



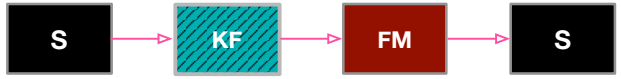
TrafficMonitoring (TM): analysis of vehicle-races in Beijing. GPS localization of the measurements, and computation of the average speed of each road of the city

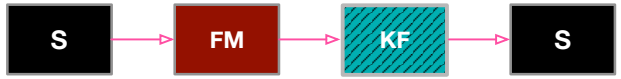


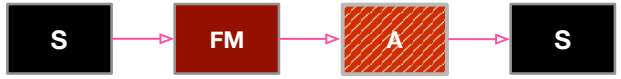
Word Count (WC): popular benchmark of streaming applications. Count the number of distinct words present in a text document (in a streamed fashion)



WindFlow Implementation

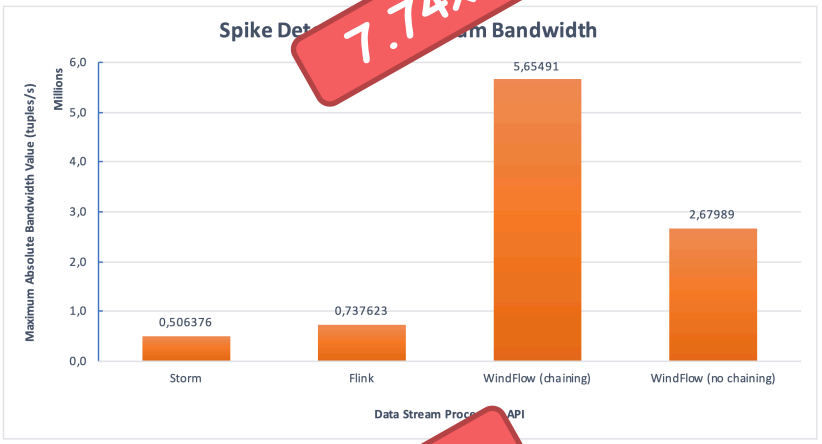


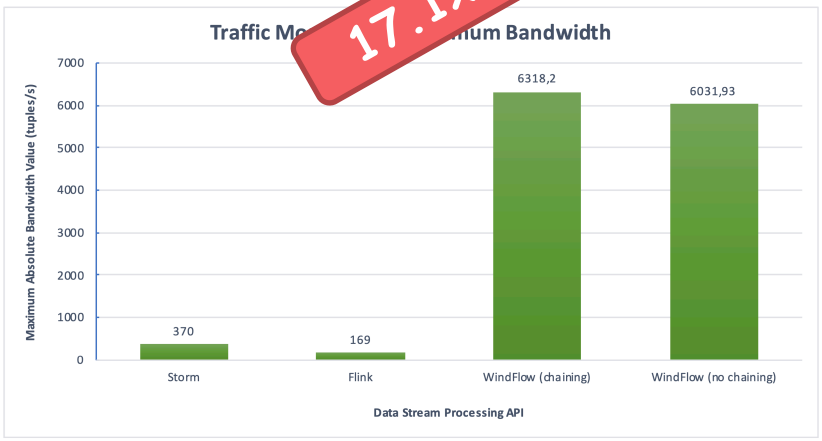


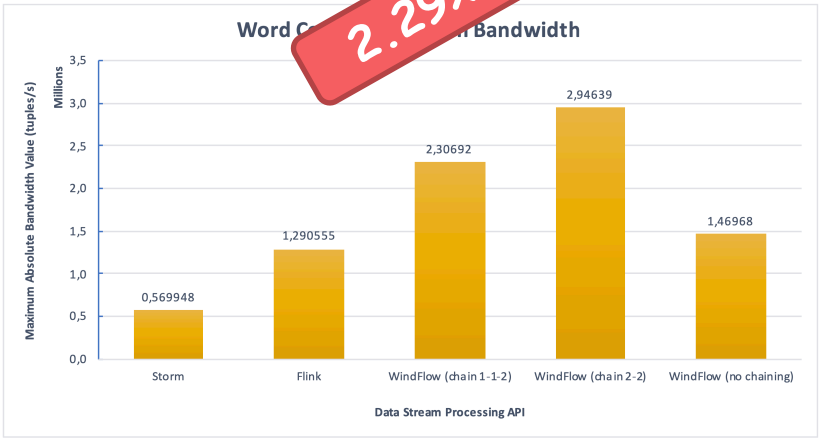


Peak Throughput

- Throughput → number of inputs processed per second in the best **configuration** (number of replicas per operator)

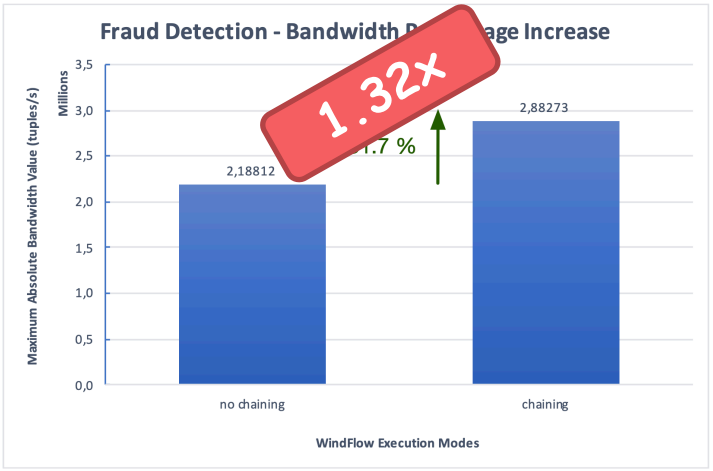




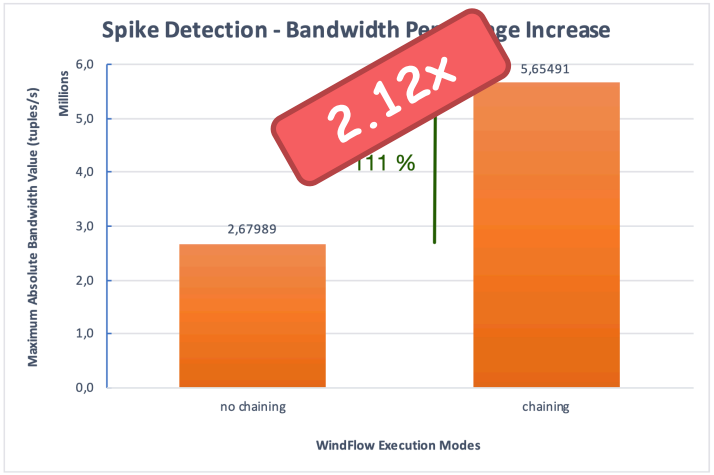


Chaining Impact

- When possible **operator chaining** reduces overheads…



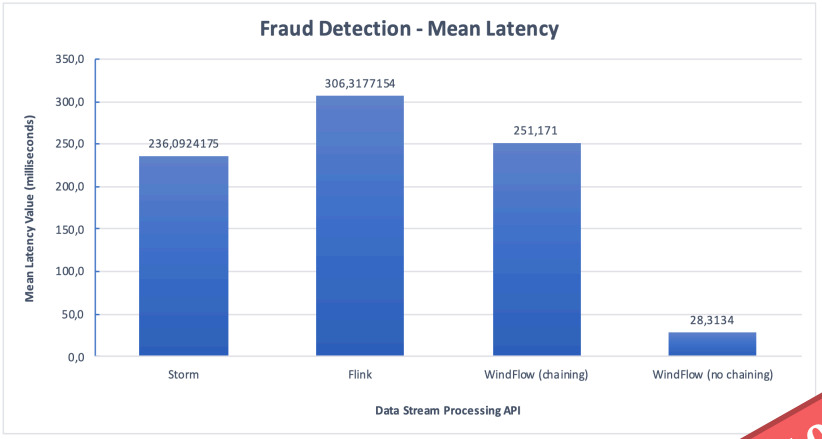
(a)

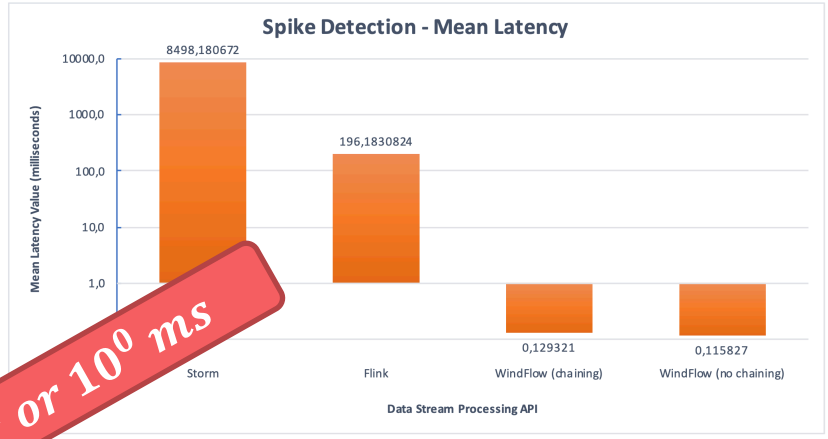


(b)

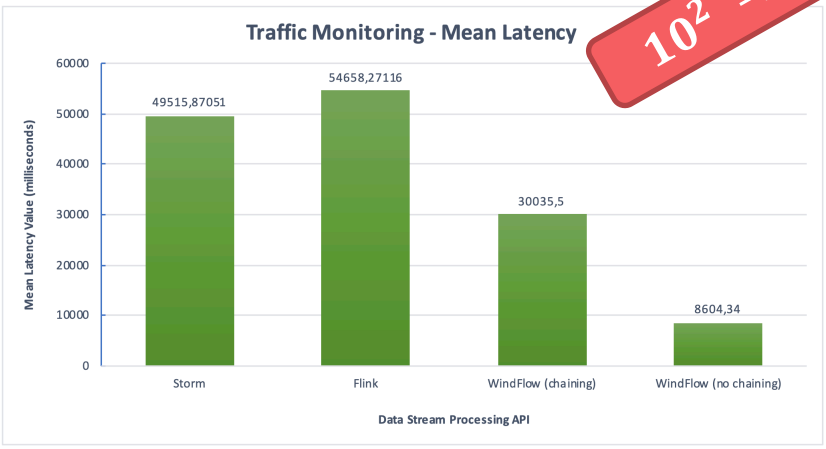
Latency

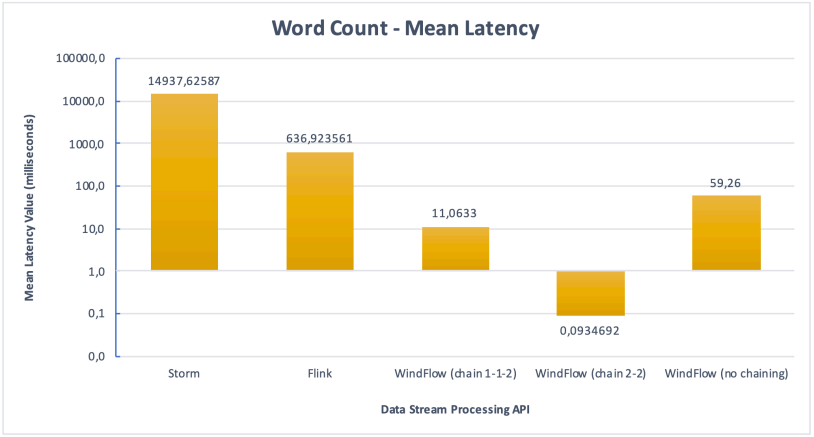
- Latency → time spent from the input arrival to when the corresponding output is produced. Below only the **mean value** for space reason….





$10^2 \rightarrow 10^1$ *or* 10^0 *ms*



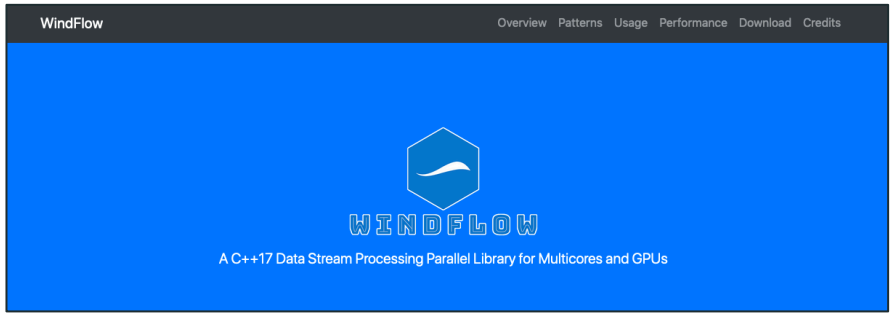


Conclusions

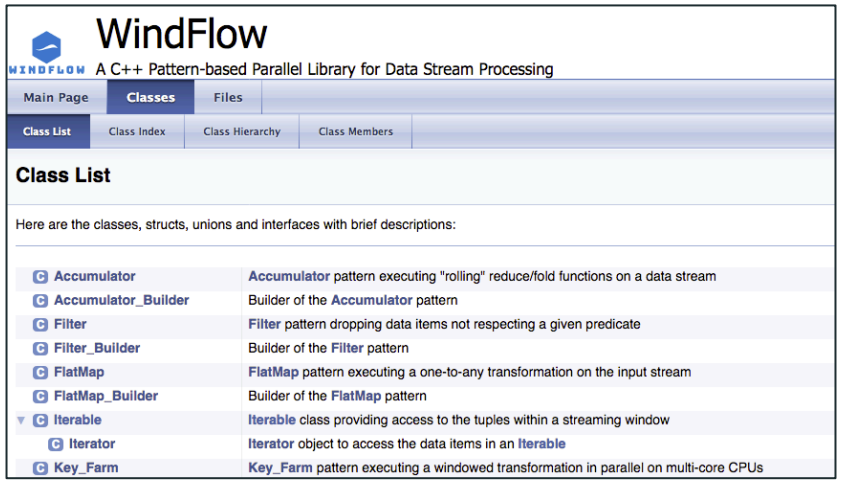
- **WindFlow** is a C++17 library for High-Throughput Low-Latency Data Stream Processing targeting Multicores

Available in GitHub
https://github.com/ParaGroup/WindFlow

Web page
https://paragroup.github.io/WindFlow/



Doxygen guide is available in the source code for using the API and its constructs



How to Cite

G. Mencagli, et. Al., **"Raising the Parallel Abstraction Level for Streaming Analytics Applications"** in *IEEE Access*, vol. 7, pp. 131944-131961, 2019.
doi: 10.1109/ACCESS.2019.2941183

Thank You!