
Introduction to FastFlow programming

— Massimo Torquati —

<massimo.torquati@unipi.it>

SPM lecture 7

Master Degree in Computer Science
Master Degree in Computer Science & Networking
University of Pisa

FastFlow Building Blocks (BBs)

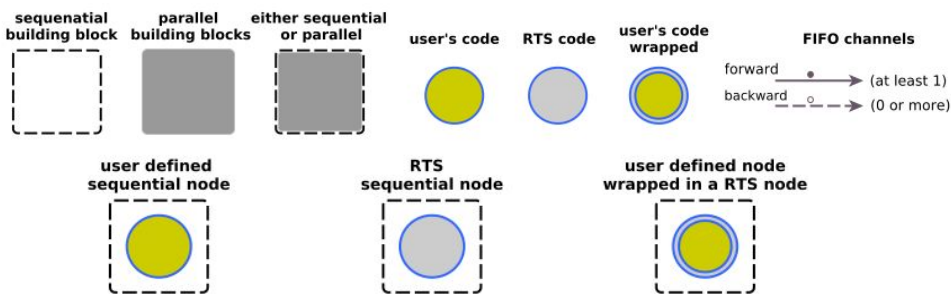
recap 1/3

- Concurrent bricks that can be used to build new high-level patterns and/or new run-time systems (RTSs) following the structured parallel programming approach
- They are like “*Lego bricks*” with different number of studs.
- **Sequential bricks:**
 - Standard node (**1-to-1**), multi-input (**N-to-1**) and multi-output (**1-to-N**) nodes
 - **combiner**: combines sequential building blocks (e.g., a multi-input and a multi-output nodes)
- **Parallel bricks:**
 - *Pipeline* (*ff_pipeline*)
 - *Farm* (*ff_farm*)
 - *All-to-All* (*ff_a2a*)

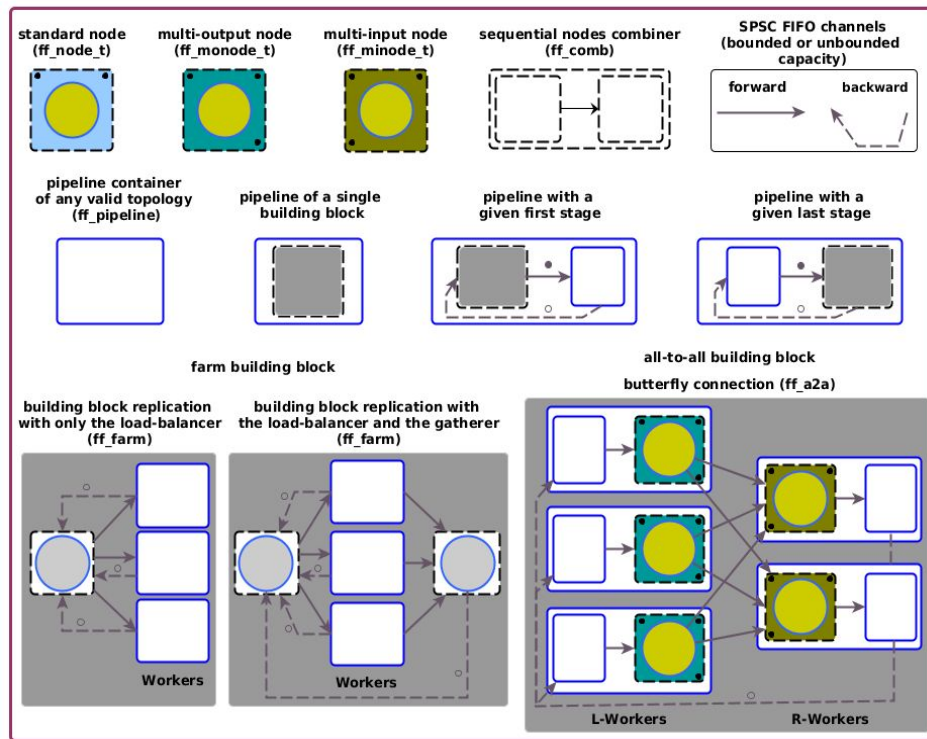
FastFlow Building Blocks Representation

2/3

Symbols used:



- **channels** are SPSC FIFO queues (default configuration).
Capacity: bounded/unbounded
Concurrency-Control: blocking/non-blocking
- **combiner**: aims at decoupling the node abstraction from its concrete implementation, useful to reuse portion of code and to reduce the number of threads used. The semantics is the one of sequential composition applied to the three service methods



Parallel Building Blocks

3/3

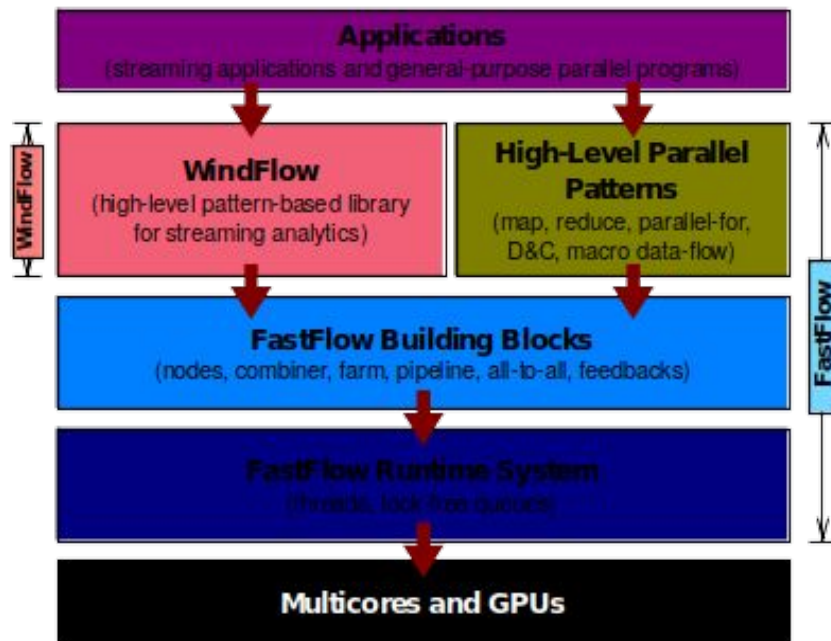
- The *ff_pipeline* models the pipeline composition of building blocks (included the pipeline itself). It is used both as a container of building blocks connected in pipeline and as application topology builder.
- The *ff_a2a* (A2A) defines two distinct sets of Workers (i.e. pipeline containers) connected according to the *shuffle* communication pattern. From the concurrency standpoint, the semantics of the A2A building block is that of two farms connected in pipeline, the first running the L-Workers and the second running the R-Workers.
 - The only requirement of the A2A building block is that the last stage of the pipeline implementing the generic L-Worker must be a multi-output node whereas the first stage of the pipeline implementing the generic R-Worker must be a multi-input node.
- The *ff_farm* models functional replication of Workers (pipelines) coordinated by one or two master nodes (Emitter and Collector, which are special kind of multi-output and multi-input nodes). Emitter and Collector nodes can be user-defined

What we can do with the Building Blocks?

- To show the flexibility and the power of the FastFlow Building Blocks, we briefly describe the WindFlow library which has been built on top of them
- Specifically, the WindFlow library leverages
 - Sequential and parallel Building Blocks
 - Concurrency graph transformer, i.e., the utility functions provided by the library to reshape the graph of nodes
- **More info:** <https://paragroup.github.io/WindFlow/>

A streaming DSL built on top of Building Blocks





- **WindFlow** a Data Stream Processing C++17 embedded DSL
 - It currently targets multi-cores.
 - It provides support for GPUs, too
- WindFlow provides the user with a *compositional interface* (fluent style) of basic and complex streaming operators
 - We will have a quick look only at the basic ones




WindFlow “basic” operators

- Stream generation (source operator)
- How is the stream transformed?
- Stream gathering (sink operator)





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

Basic Operators	
	Map operator applies a one-to-one transformation producing one output per input
	Filter operator applies a boolean predicate on each input item and drops the ones returning false
	FlatMap operator producing one or more output items per input item consumed
	Accumulator operator executes a “rolling” reduce or fold function on the inputs partitioned by key

Sink	
	Sink operator is in charge of absorbing the input stream of items

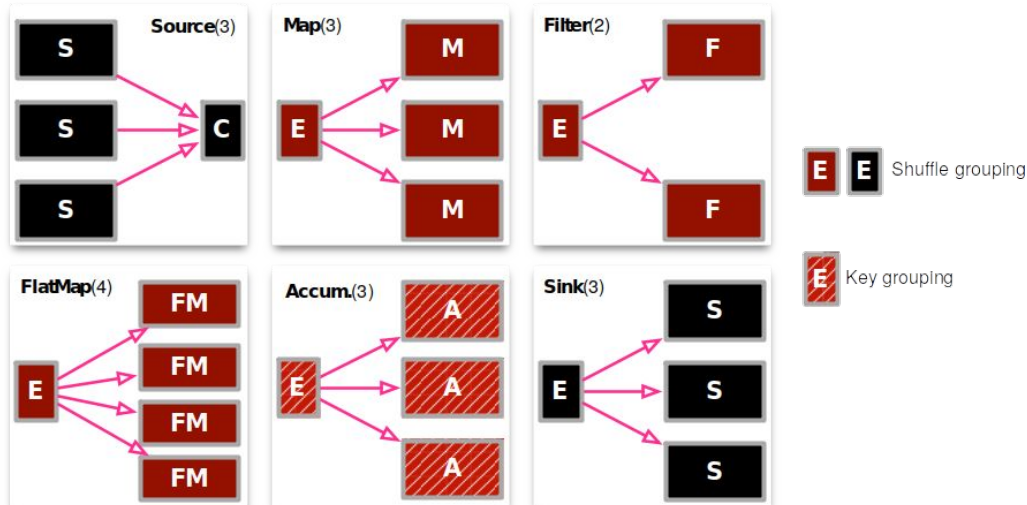
WindFlow “complex” operators

- Operators working on windowed queries (complex composition of farm and A2A)

Window-based Operators		
T1* →		→ 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* →		→ T2*
Windowed Farm in charge of executing a windowed query in parallel on distinct streaming windows (key grouping is not required for parallelism)		
T1* →		→ T2*
Paned Farm in charge of executing window-based operators in parallel by exploiting window overlapping (key grouping is not required for parallelism)		
T1* →		→ 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)		

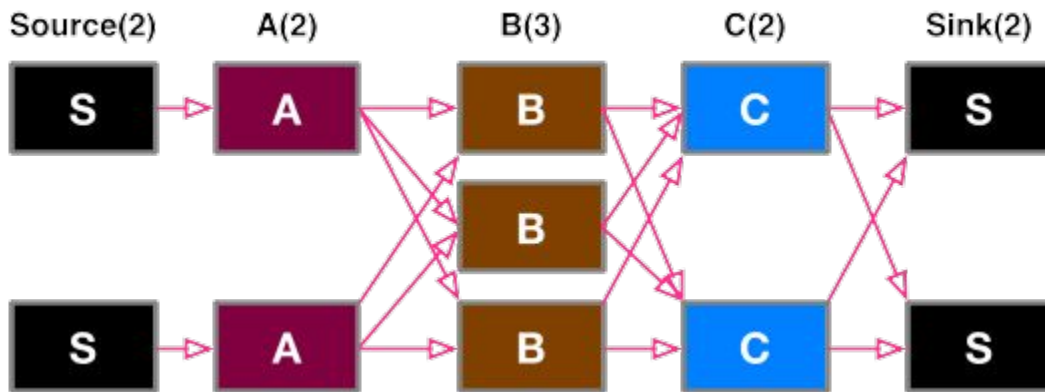
FastFlow representation of basic operators

- Source, Map, Filter, FlatMap, Accumulator and Sink are implemented with the FastFlow's All-to-All (A2A) building block
- The A2A building block is used as a sort of “container” of
 - the set of nodes in charge of executing the operator's function in parallel. The cardinality depends on the parallelism attribute provided using the *withParallelism()* method of the API
 - the distribution/collection nodes for routing stream items to/from the replicas of the operator



Creating the streaming network

- WindFlow provides a specific construct to compose patterns instances and build the network
- This construct is called **MultiPipe**. It is composed by
 - multiple pipelines working in parallel (each pipeline is a linear chain of replicas of the utilized operators)
 - pipelines are not independent but data items may cross a pipeline and jump to another one



Two connection modes:

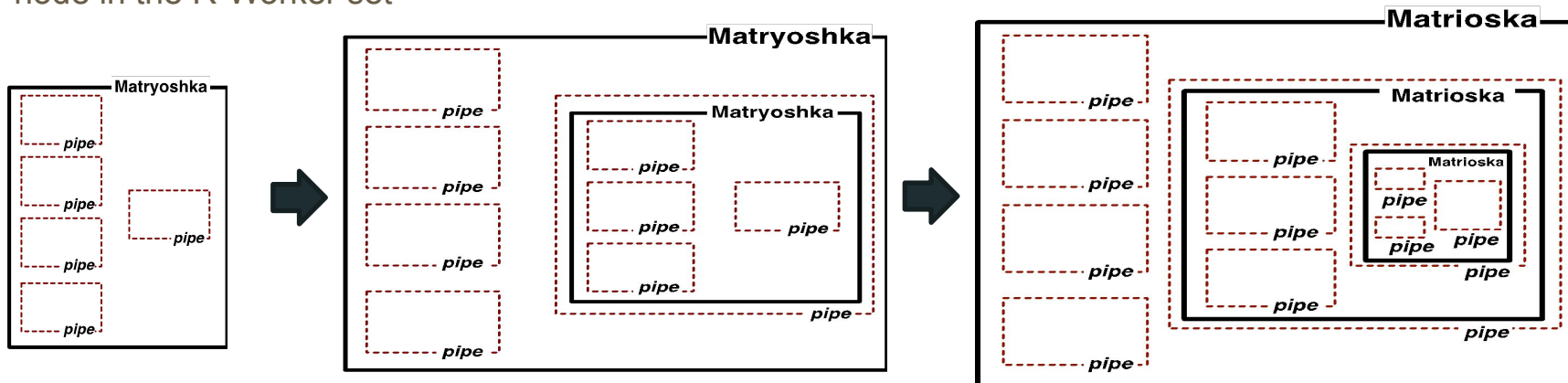
Direct: one replica connected to exactly one replica of the next operator in the MultiPipe

Shuffle: one replica connected to all the replicas of the next operator in the MultiPipe

- The connection mode depends on the type of patterns and their cardinalities
- Shuffle connection can be: random, key-based or window/pane-based depending

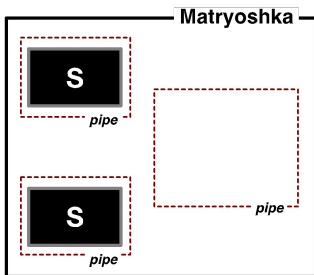
Streaming network at FastFlow level

- To build a MultiPipe structure, the WindFlow RTS uses the A2A building block in a way resembling a *Matrioshka* (this is transparent to the user)
- To add a new operator to the MultiPipe, a new Matrioshka can be instantiated and nested within the last Matrioshka of the MultiPipe
- A Matrioshka is an instance of the A2A with N pipeline nodes in the L-Worker set and one pipeline node in the R-Worker set



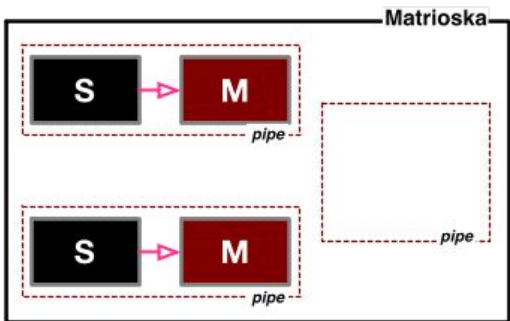
Adding operators (1)

- Starting case: a Source pattern instance with parallelism is added to an empty MultiPipe named “test1”. The initial Matryoshka is prepared with pipeline instances in the L-Worker set, each one having only one Source node inside



```
// main
int main(int argc, char *argv[]) {
    ...
    Pipe pipe("test1")
    Source_Functor G;
    Source source = Source_Builder(G).withName("test1_source")
                                   .withParallelism(2)
                                   .build();
    pipe.add(source);
}
```

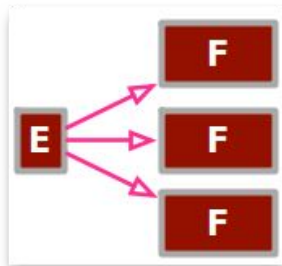
- First case: a BASIC pattern (Map, Filter, FlatMap) is added to the MultiPipe. If the parallelism degree is equal to the number of pipelines in the L-Worker set in the last Matryoshka, we proceed with direct connections



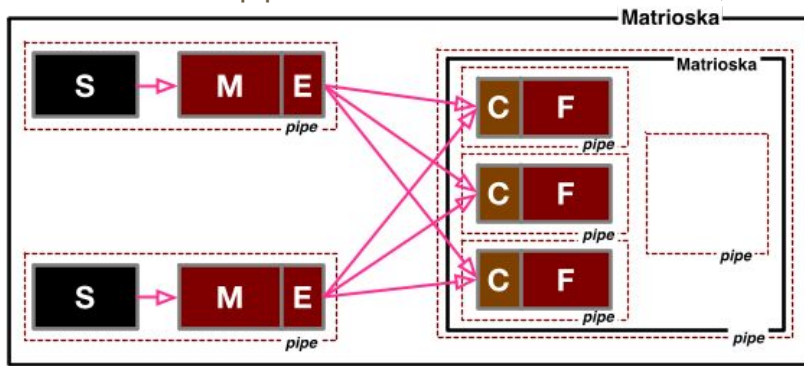
```
Map map = Map_Builder(mapF).withParallelism(2)
                        .withName("test1_map")
                        .build();
pipe.add(map);
```

Adding operators (2)

- Second case: a BASIC pattern (Map, Filter, FlatMap) is added to the MultiPipe. If its parallelism is different than the number of pipelines in the L-Worker set of the last Matrioska, we proceed with shuffling connections (the same if we introduce a key-by operator)



- The Emitter node (E) distributes input items to the replicas of the new operator. The node is replicated and sequentially combined with last node in each pipeline in the L-Worker set. Then, a new Matrioska is instantiated and nested in the R-Worker set

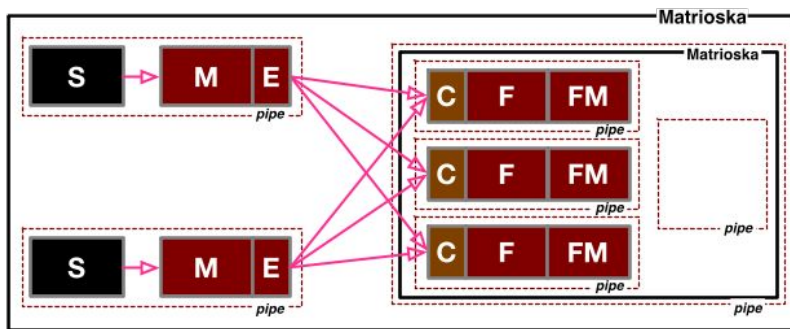


Collector node:

- FIFO ordering
- Ordering based on timestamps

Chaining operators

- Each FastFlow node is executed by a dedicated thread. To reduce the number of threads, WindFlow provides a *chaining* method to combine consecutive operators under certain conditions
- Chaining conditions:
 - The new operator is a Map, Filter, FlatMap or Sink instance
 - The number of replicas of the new operator must match the ones of the previous one



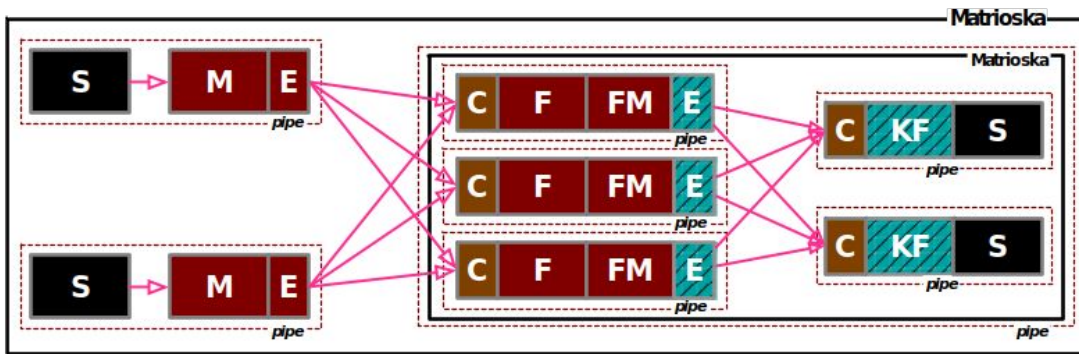
The operator chaining feature is implemented by using the **node combiner** building block

```
FlatMap flatmap = Filter_Builder(flatmapF).withParallelism(3)
    .withName("test1_flatmap")
    .build();
pipe.chain(flatmap);
```

A FlatMap instance (FM) with parallelism 3 is chained with the MultiPipe instance. The previous pattern in the MultiPipe is a Filter (F) with the same cardinality: chain is possible (otherwise the chain() performs an implicit add())

Running the network

- A MultiPipe instance is runnable if two conditions are satisfied
 - It has a Source operator (it must be the first one of the MultiPipe)
 - It must end with a Sink operator (no further operator can be added after a Sink)
- Different execution modes (synchronous and asynchronous)



- 24 FastFlow nodes (excluded combiner nodes)

- 9 Threads running the network

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

pipe.chain(sink);
pipe.run_and_wait_end(); // synchronous run of the MultiPipe
```

Throttling the number of Workers

- **Concurrency throttling** refers to the possibility to dynamically change the number of threads implementing a parallel application
- Why?
 - To adapt the number of resources used at run-time
 - To reduce power consumption if the system is not fully loaded
 - To keep-up with higher input rate than the average
 - To dynamically discover the optimal number of workers in a farm (and all-to-all)
 - etc...
- In the FastFlow case this applies to the possibility of varying the number of Workers in a farm, and potentially in a All-to-All building blocks
 - For the All-to-All, concurrency throttling mechanisms not implemented, yet!

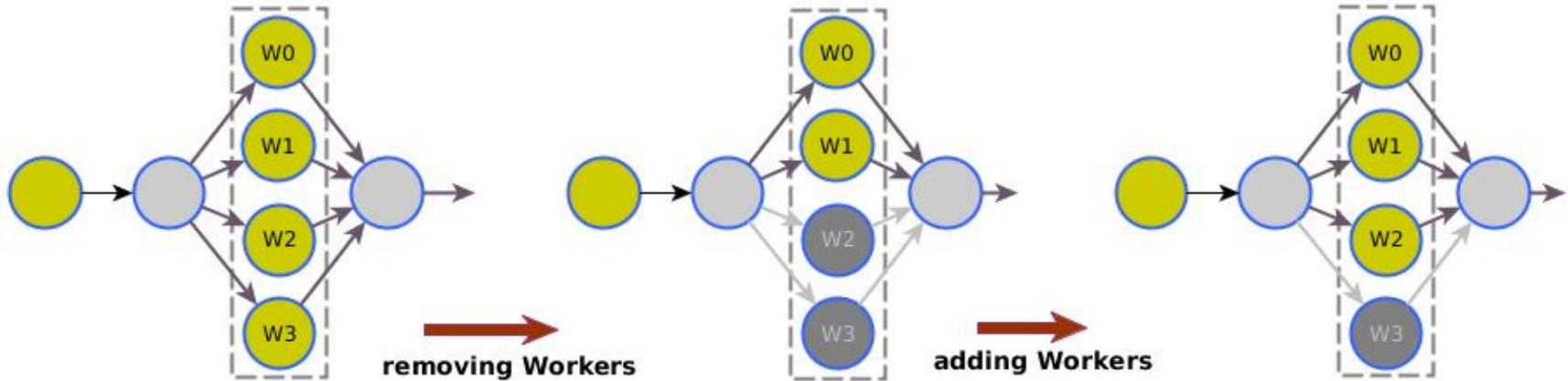


FastFlow basic mechanisms

- When a FF node receives EOS, it is propagated and the node terminates (default behavior)
- But if the thread has the freeze flag set, instead of being terminated the thread *is put to sleep*
 - The EOS is propagated as well!
 - Before going to sleep, the node executes `svc_end`
- Afterwards, the node can be woken-up by sending an explicit signal to the sleeping thread
- When the thread re-starts, it executes again `svc_init`
- The EOS is propagated throughout the pipeline as any “standard” stream item, therefore it is difficult to manage node freezing if we just want to “freeze” a portion of the graph (e.g. farm’s Workers)
- The special message is **EOSW**, which is propagated only inside a farm ($E \rightarrow Ws \rightarrow C$)
- Another special message **GO_OUT** that is not propagated (eosnotify is not called)

Throttling the number of Workers in a farm

- farm building block

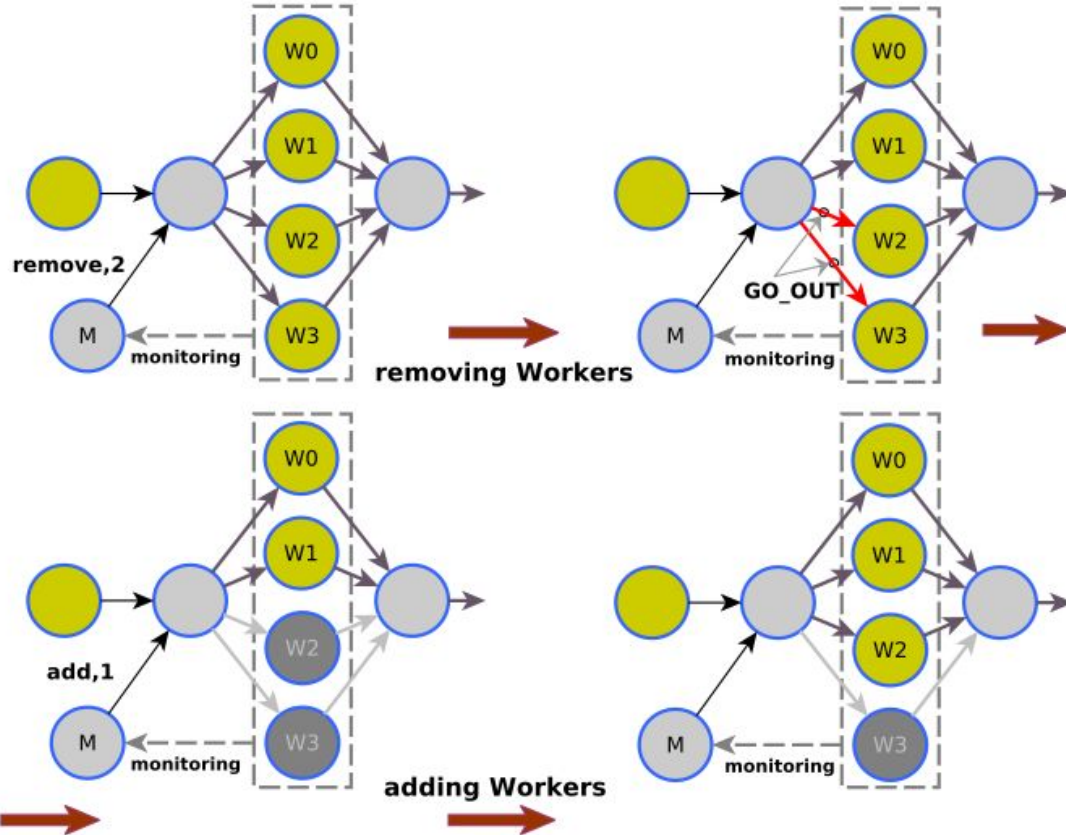


- Threads associated to nodes are **not** created and destroyed dynamically
- Instead, they are stopped (put to sleep) and restarted (woken-up) on the fly

Simple examples

- How to set the freeze flag?
 1. By starting the FF graph with the ***run_then_freeze*** (instead of *run/run_and_wait_end*) and then ***wait_freezing*** (instead of *wait*)
 2. or by calling *freeze* after the thread associated to the node has already started
- How to restart a sleeping node?
 - By using the method ***thaw*** provided by the farm's Emitter and by the 1-to-N node (*ff_monode*)
- Let's have a look at a simple examples
 - `farm_square1_throttling.cpp`

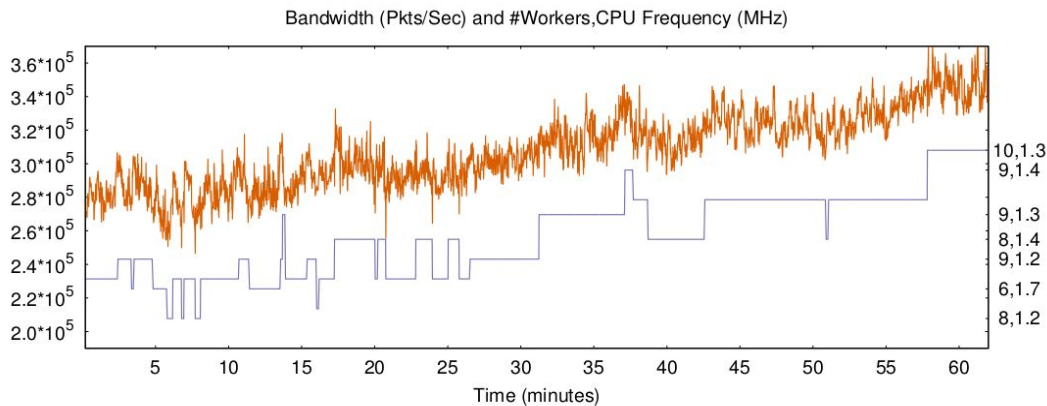
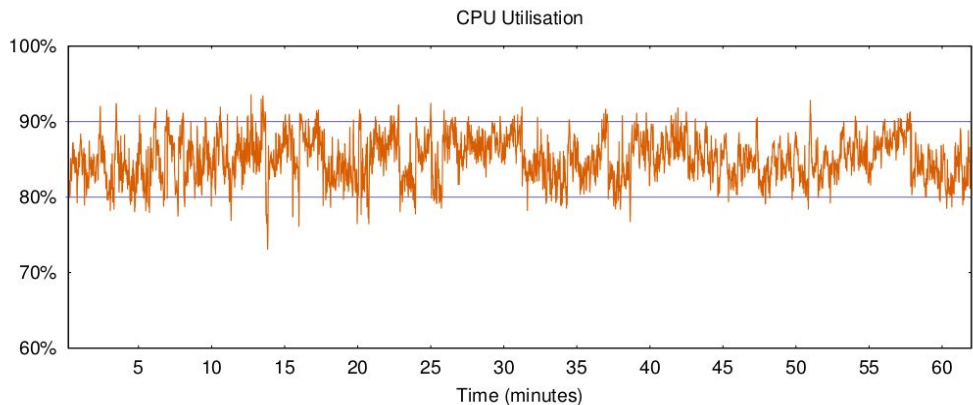
ff_farm with the manager node



FastFlow examples with a manager node

- Let's have a look at the examples that uses a very simple manager node (both examples contained in the *test* directory within the FastFlow top-level folder):
 - `test_multi_output6.cpp`
 - `test_multi_output5.cpp`

Throttling Workers and Frequency scaling

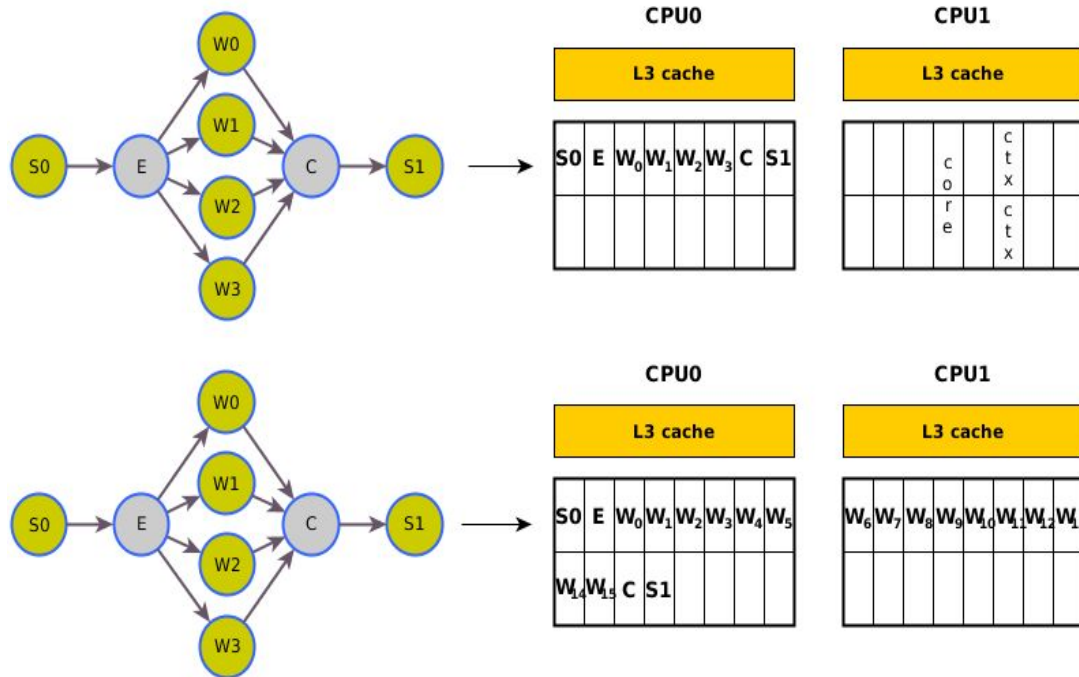


Processor affinity (CPU affinity)

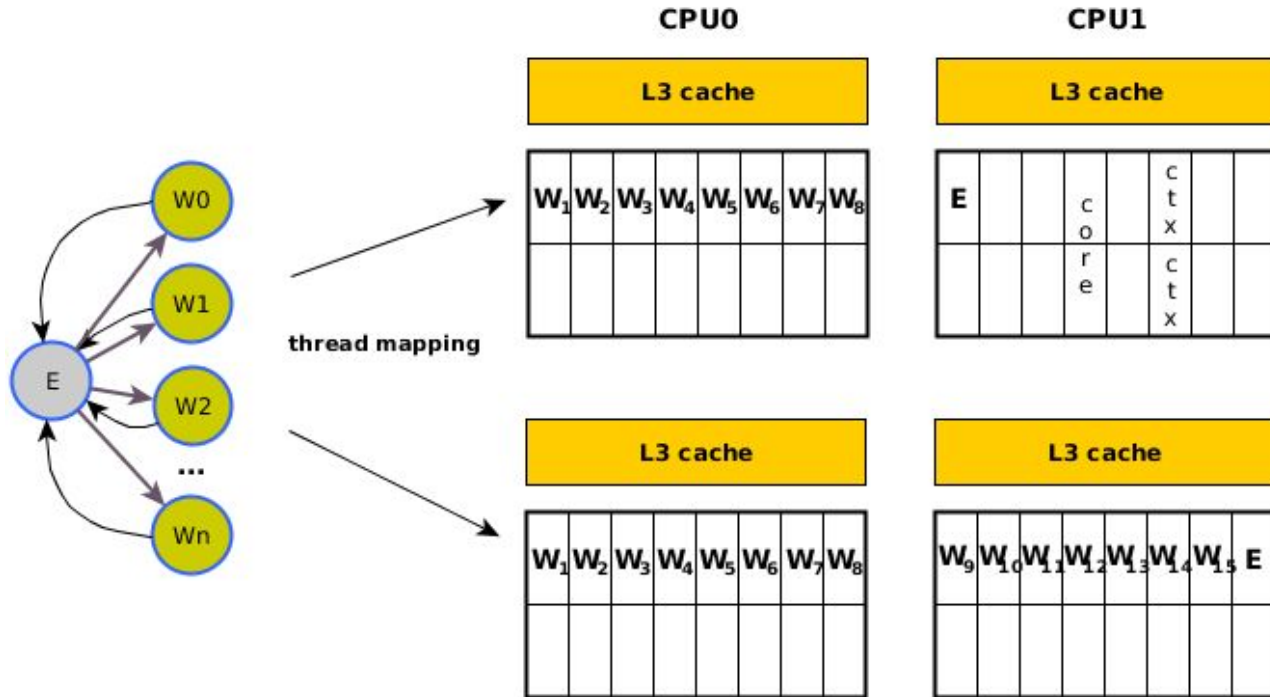
- It is a feature provided by several OS to control threads (processes) placement onto CPU cores
- The placement of threads onto cores is also called *thread mapping* (also *thread pinning*)
- Deciding the optimal mapping of threads to core is NP-hard
- There are several heuristics that work well in practice, which are implemented by the OS
- If the platform and the application structure is known in advance, it is possible to derive good mapping heuristics

Threads mapping/pinning in FastFlow

- Default threads mapping: **linear**. Depth-first visit of the graph.
- Threads are pinned to sibling cores. Good heuristic in the Producer-Consumer model
- However, that it is not always a good choice: if the working set does not fit in the last-level cache, the performance could be worse
- The **NO_DEFAULT_MAPPING** compilation flag disables the default thread mapping
- It is possible to map threads in the `svc_init()` “by hand”
- The mapping could be managed by using the **optimize_static** function or by using the **no_mapping()** method offered by the top-level farm, A2A and pipeline BBs. More options are coming!



Controlling threads mapping in FastFlow



- Let's have a look at the example *farm_pinning.cpp* showing some mechanisms

Concurrency Control

- We already know that a communication channel connecting two FastFlow nodes is implemented by using a **non-blocking** FIFO queue (Single-Producer Single-Consumer)
 - Non-blocking is the default concurrency control mode (see SPM-lecture 1)
- If the number of threads used is greater than the number of cores, the busy-waiting loop executed by the threads might introduce too much noise for other threads
 - Suppose a collector node that “sporadically” receive data elements that share the core with a Worker (or worse with the Emitter) in a farm
- In these cases, a **blocking** concurrency control mode for accessing the queues might be more appropriate even though it is intrinsically less reactive (it has more overhead)
- In FastFlow it is possible to change to the blocking mode system-wide by compiling the code with **-DBLOCKING_MODE**

Concurrency Control

- It is also possible (as for thread mapping) to switch to blocking mode at run-time by using **optimize-static**
- Example (see, for example, *test_optimize.cpp*):

```
OptLevel opt;  
opt.max_mapped_threads=ff_realNumCores();  
opt.no_default_mapping=true; // disable mapping if #threads > max_mapped_threads  
opt.max_nb_threads=ff_realNumCores();  
opt.blocking_mode=true; // enable blocking mode if #threads > max_nb_threads  
optimize_static(myPipe, opt);
```