

---

---

# FastFlow Building Blocks

— Massimo Torquati —  
<[torquati@di.unipi.it](mailto:torquati@di.unipi.it)>  
Pisa, 2019

---

---

# High-level Parallel Patterns

- Pattern-based Parallel Programming is nowadays a **well-recognized approach for raising the level of abstraction in parallel computing** without sacrificing performance and performance portability features.

## Notable Parallel Patterns

### Data parallel

- Map
- Reduce
- Stencil
- Mapreduce
- Divide&Conquer

### Stream parallel

- Pipeline
- Farm
- Iteration
- Filter

### Higher level

- Divide & conquer
- Convolution
- Pool evolution
- Data Parallel Stream  
Pattern (windowed)

### Control parallel

- Conditional
- Speculative
- DAG

# Are they enough? Do we have all we need?



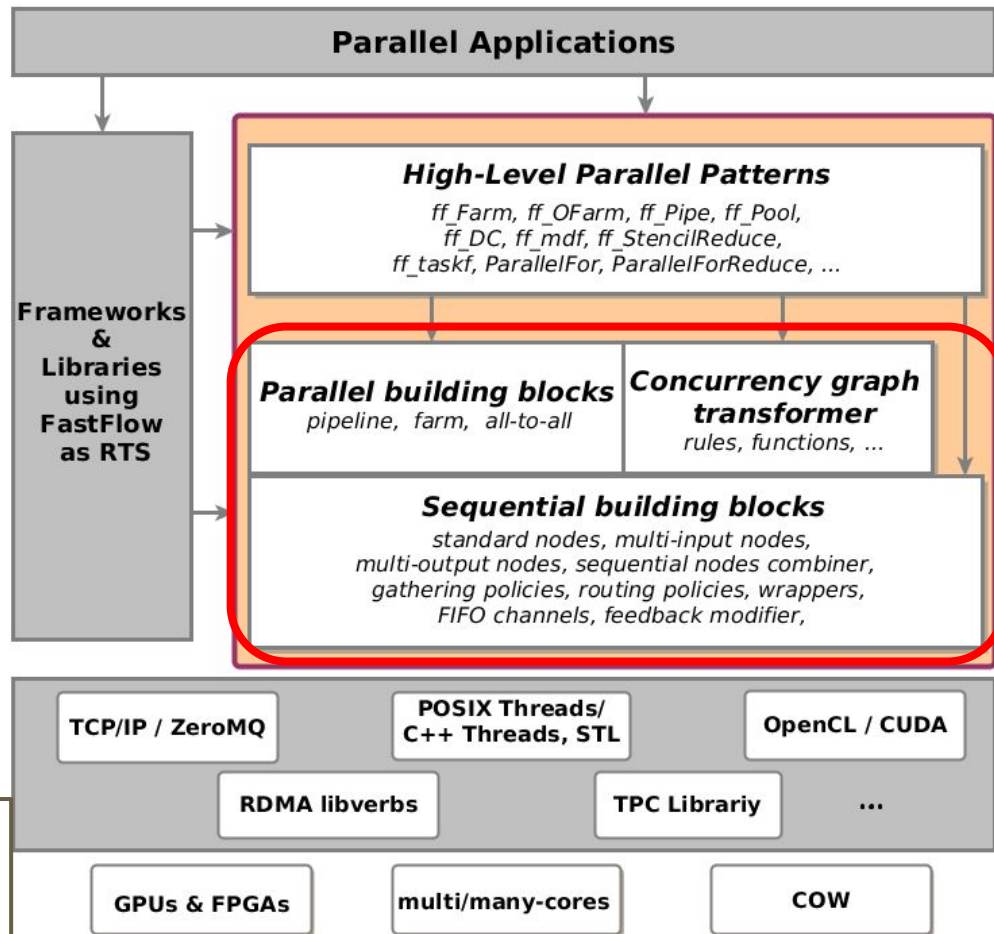
**Do it yourself!**

By combining and customizing **core patterns** and **basic mechanisms**

By using the **structured parallel programming methodology** even for building high-level parallel patterns as well as Run-Time Systems (RTSs)

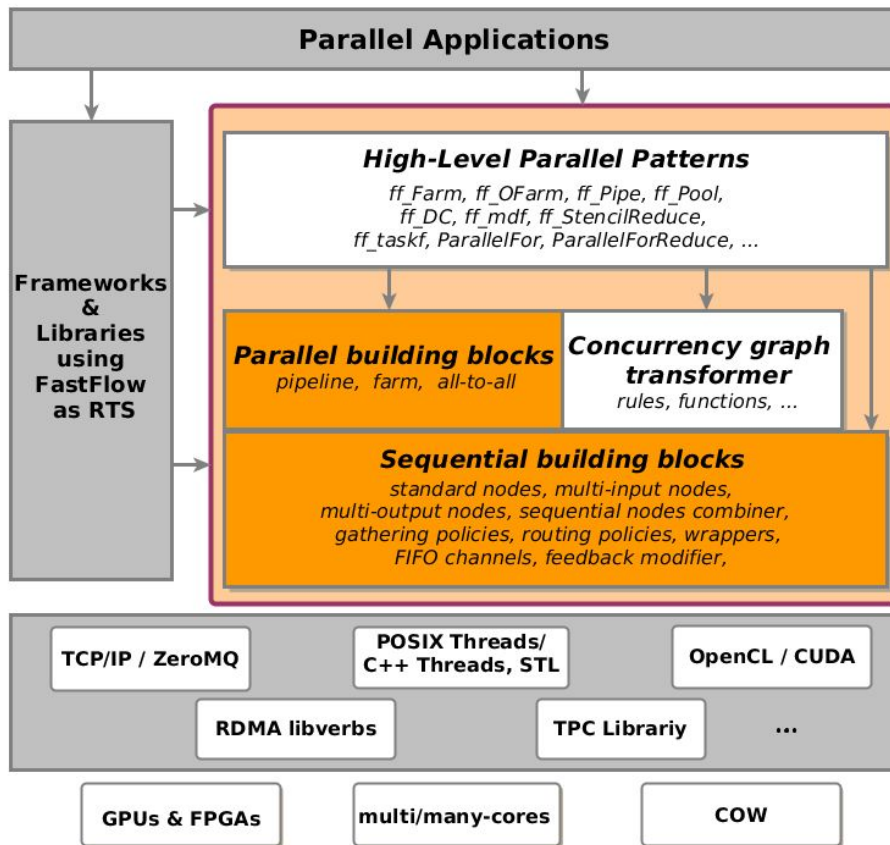
# FastFlow 3.0 stack

- High-level Parallel Patterns
- Parallel and Sequential Building Blocks
- Concurrency graph transformer



M. Torquati. "Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns." Ph.D. Thesis in Computer Science, 2019

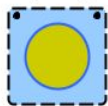
# FastFlow Building Block layer



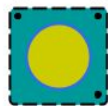
# FastFlow Building Blocks

sequential nodes

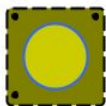
standard node  
(ff\_node\_t)



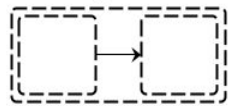
multi-output node  
(ff\_monode\_t)



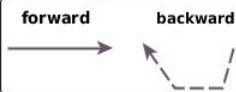
multi-input node  
(ff\_minode\_t)



sequential nodes combiner  
(ff\_comb)



SPSC FIFO channels  
(bounded or unbounded capacity)

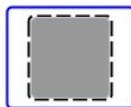


channels

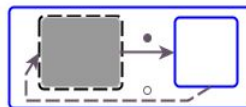
pipeline container  
of any valid topology  
(ff\_pipeline)



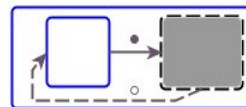
pipeline of a single  
building block



pipeline with a  
given first stage



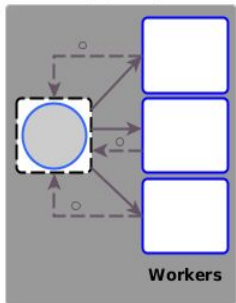
pipeline with a  
given last stage



Pipeline composition

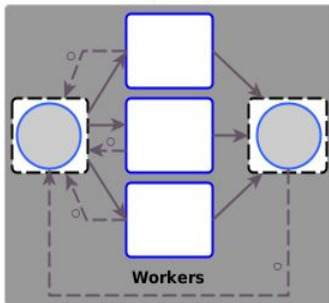
farm building block

building block replication  
with only the load-balancer  
(ff\_farm)



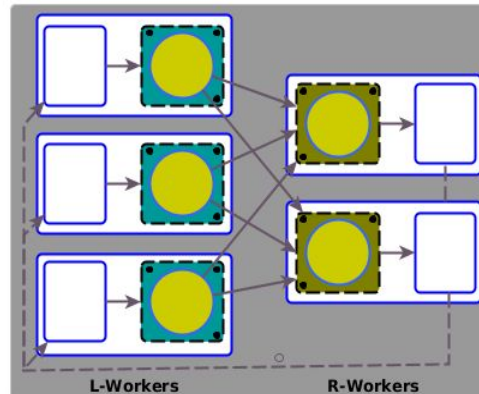
Workers

building block replication  
with the load-balancer and the gatherer  
(ff\_farm)



Workers

all-to-all building block  
butterfly connection (ff\_a2a)



L-Workers

R-Workers

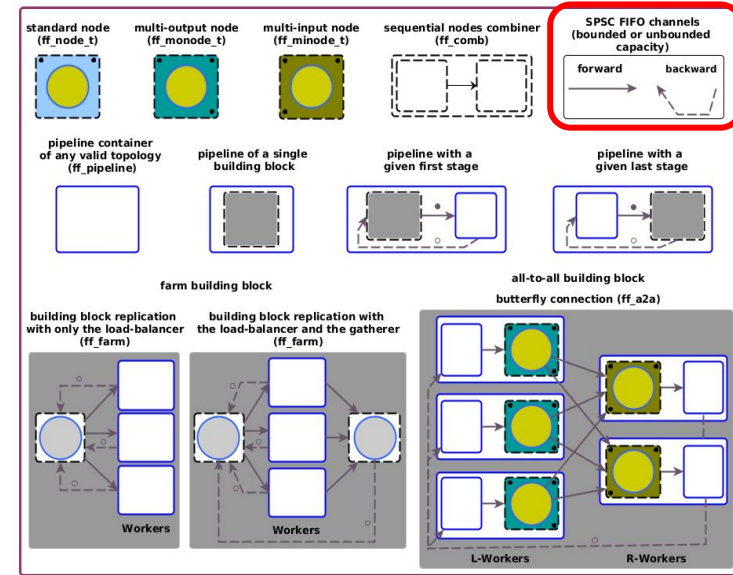
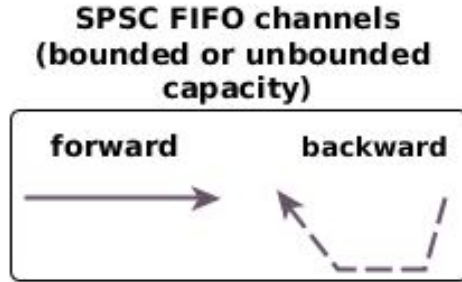
Parallel nodes

# Why Building Blocks?

- Promote **LEGO-style parallel programming**
- They have a clear functional and parallel semantics
- Promote **Performance Portability**
  - they may have different implementations for different platforms
- They “simplify the life” of the RTS programmer
  - ... while Parallel Patterns mainly “simplify the life” of Application Programmers
- Simple/Efficient **Producer-Consumer Data-Flow semantics**
  - PC semantics enables memory/cache optimizations
- The concurrency graph composed by building blocks can be ***decomposed and re-assembled in different ways enabling performance optimizations and increasing flexibility***
  - *chaining/fusion, fission, pipeline composition*

Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. “**Design patterns percolating to parallel programming framework implementation.**” International Journal of Parallel Programming, 2014.

# FastFlow Building Blocks

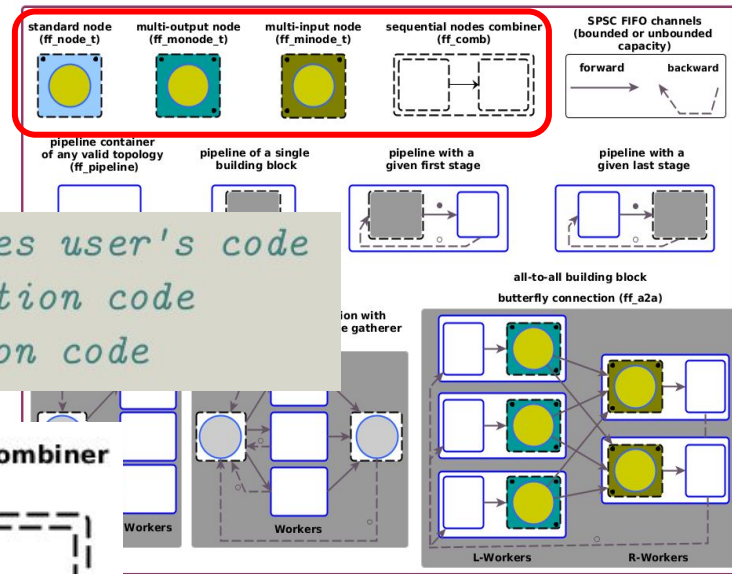


- Channels built on top of a lock-free SPSC FIFO queue
- Bounded and unbounded capacity (configurable)
  - Backward channels have always unbounded capacity
- Concurrency control: blocking and non-blocking (automatic experimental)

# FastFlow Building Blocks

```

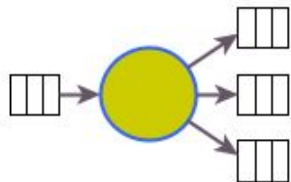
virtual TOUT* svc(TIN* task) = 0; // encapsulates user's code
virtual int   svc_init();        // initialization code
virtual void  svc_end();         // finalization code
    
```



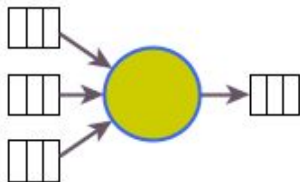
standard node (ff\_node\_t)



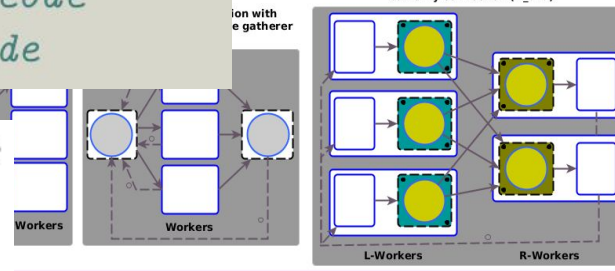
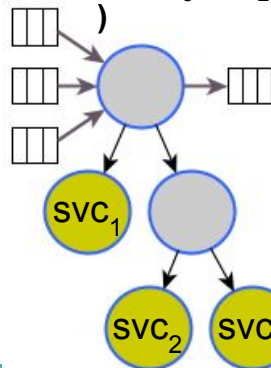
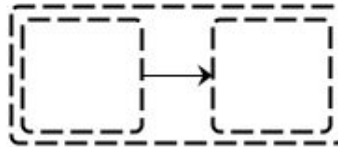
multi-output node (ff\_monode\_t)



multi-input node (ff\_minode\_t)

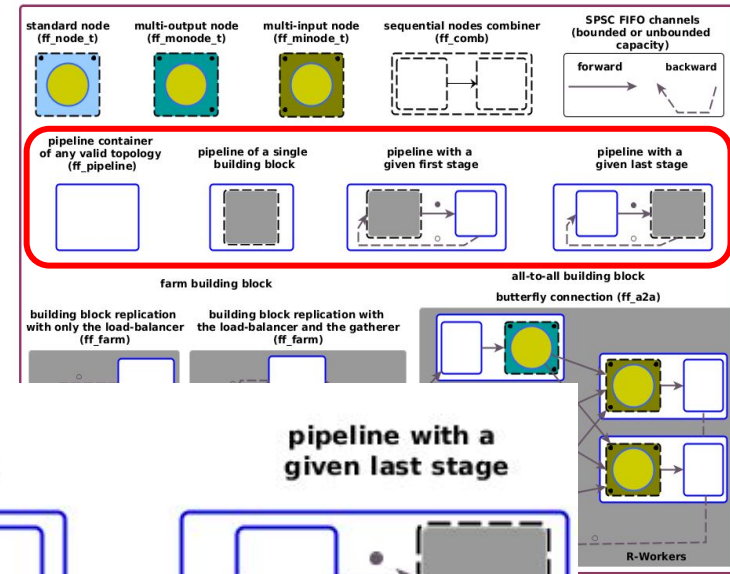


sequential nodes combiner (ff\_comb)



# FastFlow Building Blocks

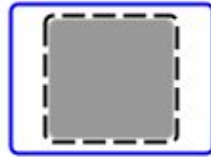
1. Building blocks' container
2. Data-Flow pipeline executor



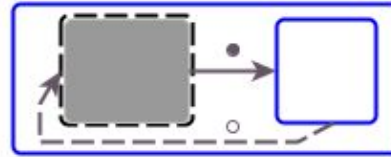
pipeline container of any valid topology (ff\_pipeline)



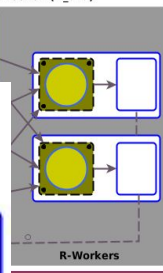
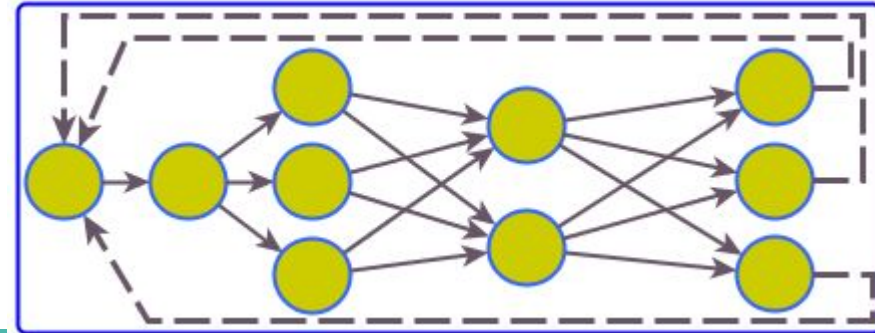
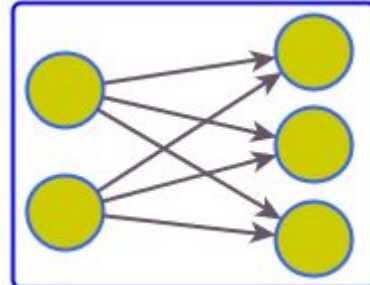
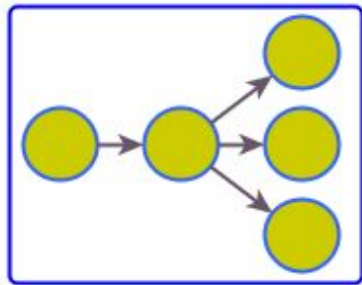
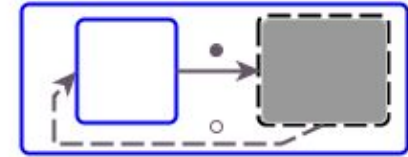
pipeline of a single building block



pipeline with a given first stage



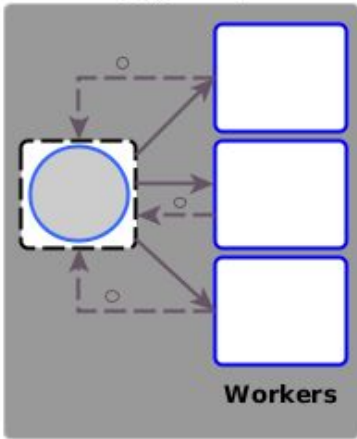
pipeline with a given last stage



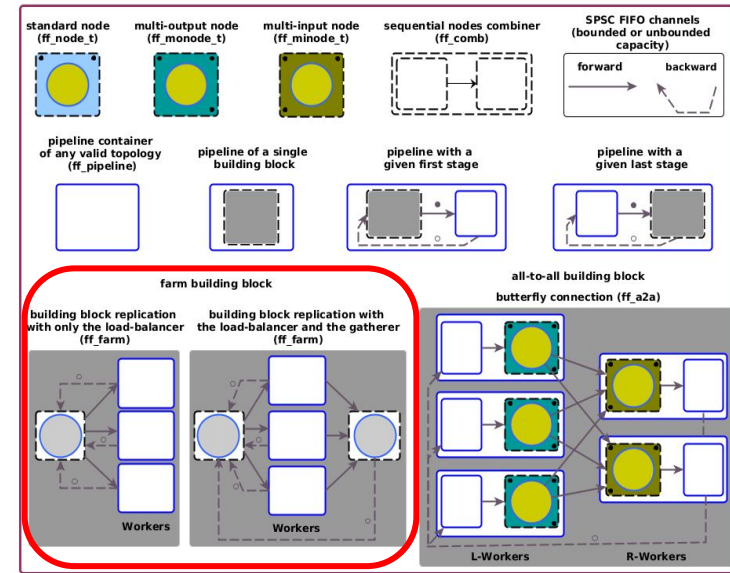
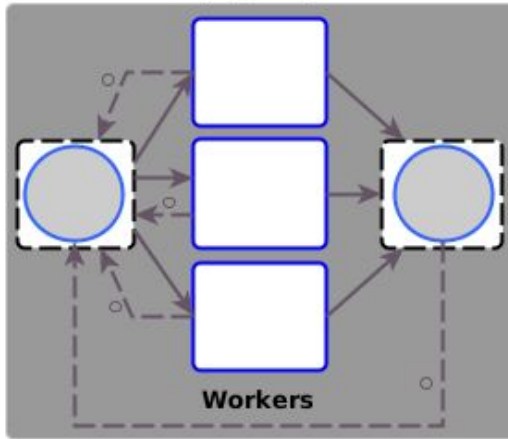
# FastFlow Building Blocks

## farm building block

building block replication  
with only the load-balancer  
(ff\_farm)

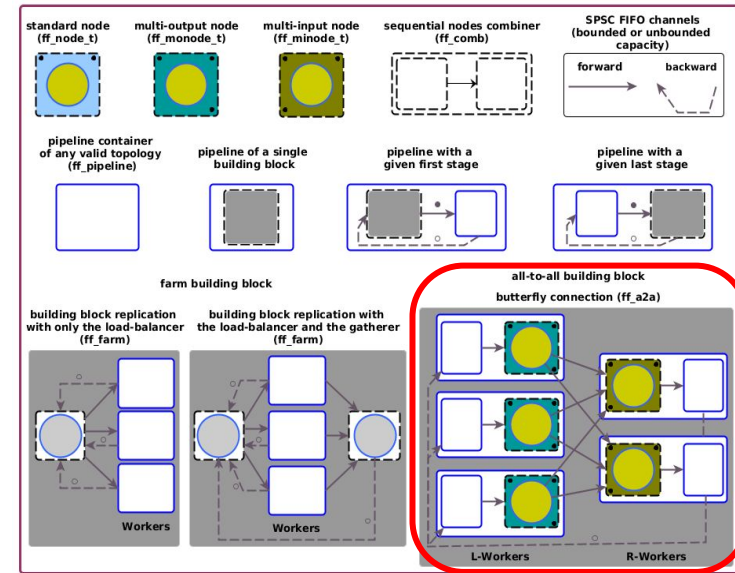
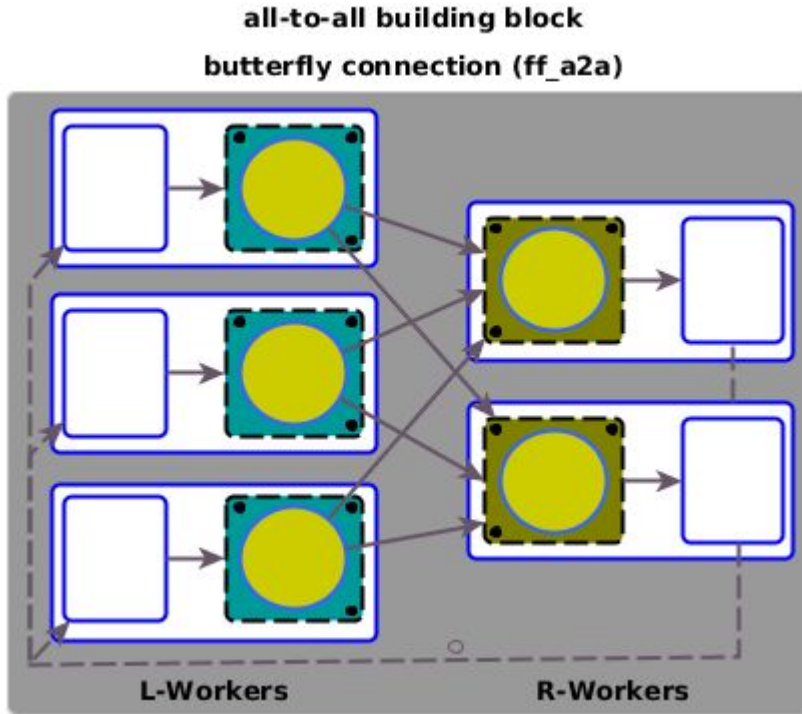


building block replication with  
the load-balancer and the gatherer  
(ff\_farm)



- Master-worker model variants
- Configurable and user-definable scheduling and gathering policies
- Concurrency throttling (extra-functional feature)
- Feedback channel modifier (optional)

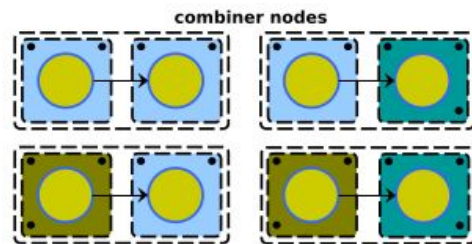
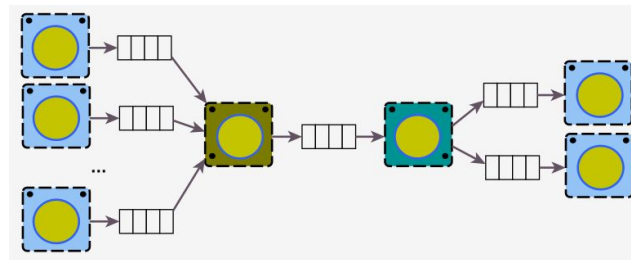
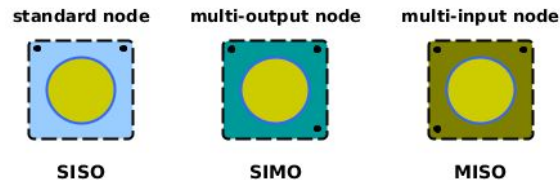
# FastFlow Building Blocks



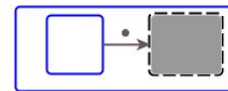
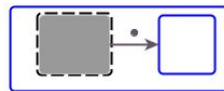
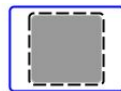
- Multi functional replication
- No master node
- Shuffle communication pattern
- User-definable scheduling and gathering policies
- Feedback channel (optional)

# Sequential BBs recap

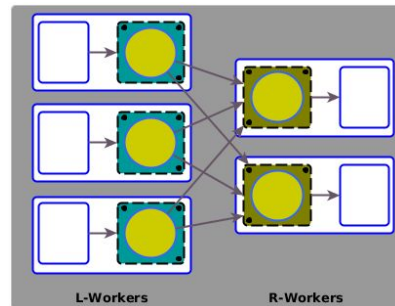
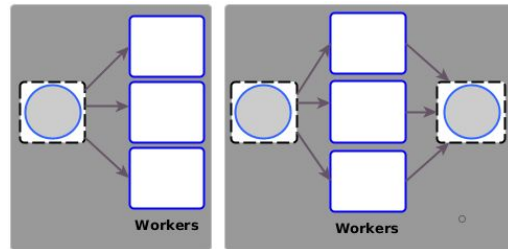
- Classified according their input/output cardinality:
  - **SISO, MISO, SIMO**
- The cardinality of MISO, SIMO is defined when they are connected to other nodes
- A connection is a FIFO SPSC channel
  - Blocking and non-blocking concurrency control
- Sequential nodes can be combined. The aim is to decouple nodes from their concrete implementations (i.e. threads)



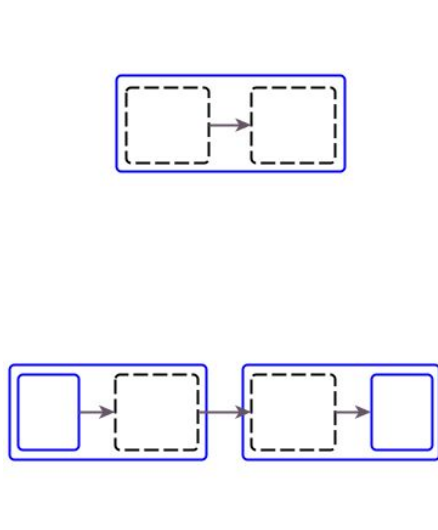
# Parallel BBs recap



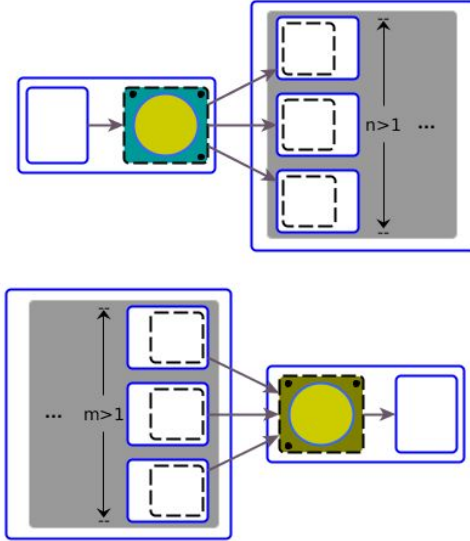
- **pipeline**: it models the pipeline composition of BBs and it also acts as a container of BBs, thus enabling BBs nesting
- **farm**: it models functional replications of pipelines coordinated by one or two master nodes (called Emitter and Collector)
- **all-to-all (A2A)**: it models two distinct sets of pipelines (L-Workers and R-Workers) connected according to the shuffle communication pattern.
  - The L-Worker pipelines must terminate with a SIMO node
  - The R-Worker pipelines must start with a MISO node



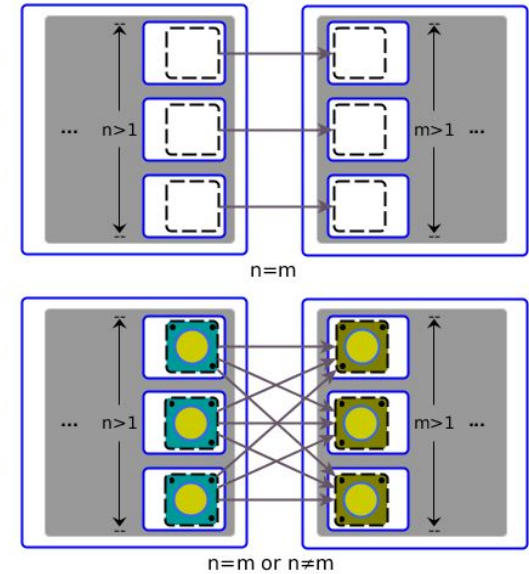
# Composition Rules



**Rule1**

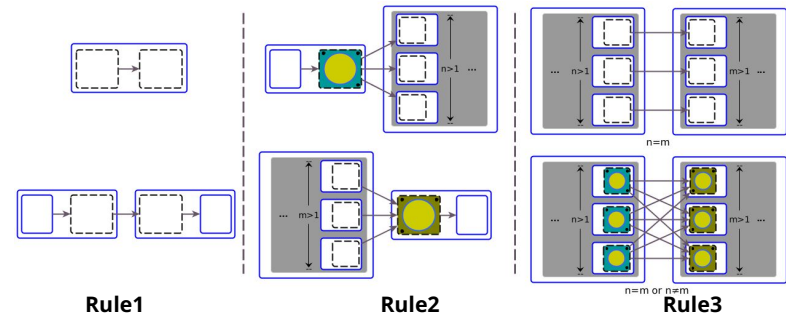


**Rule2**



**Rule3**

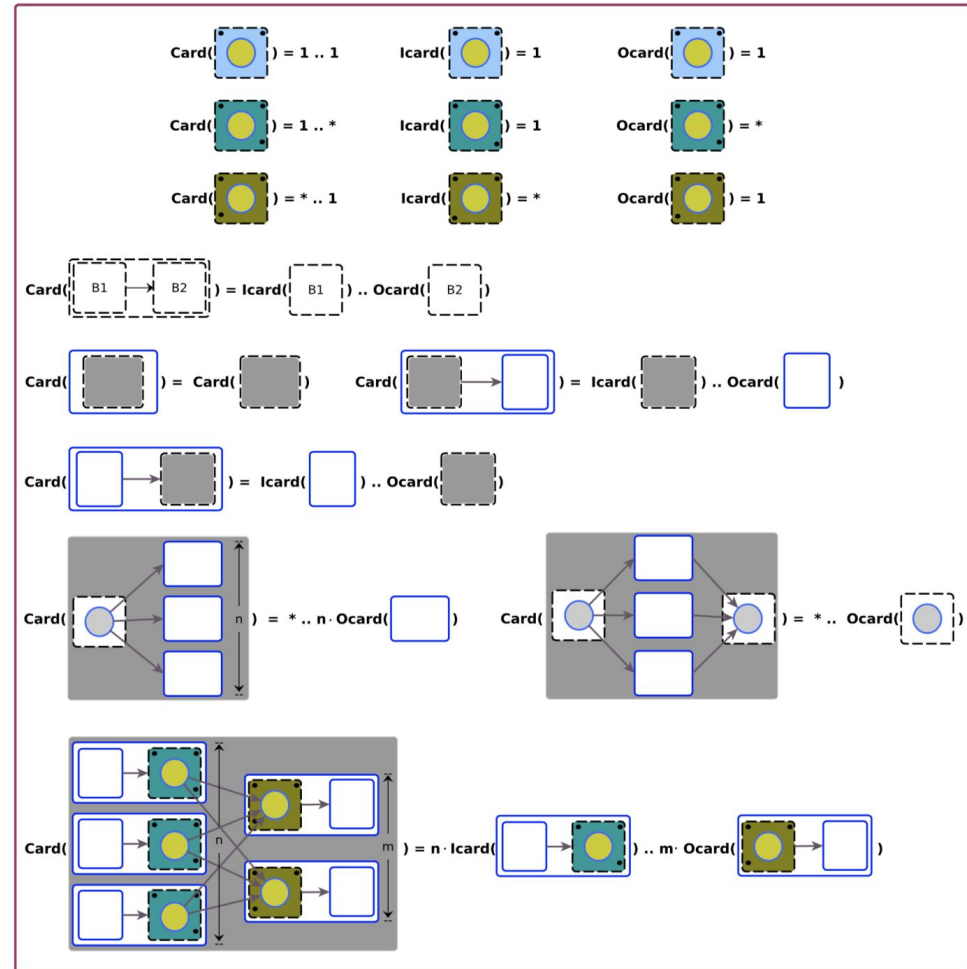
# Composition Rules



- **Rule1:** two sequential building blocks can be connected in pipeline regardless their input/output cardinality
- **Rule2:** a parallel building block can be connected to a sequential building block through a multi-output (SIMO) and multi-input (MISO) node
- **Rule3:** two parallel building block can be connected either if they have the same number of nodes regardless their input/output cardinality or through multi-input/multi-output nodes if they have a different number of nodes
  - All-to-all introduction rule between two pipelines

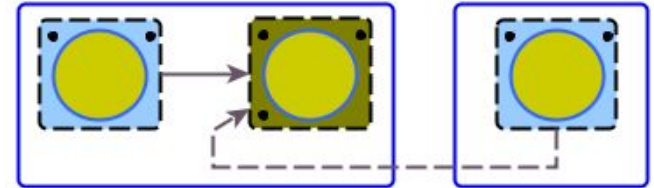
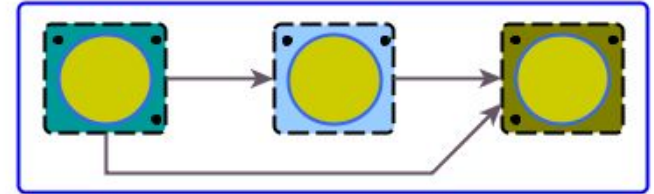
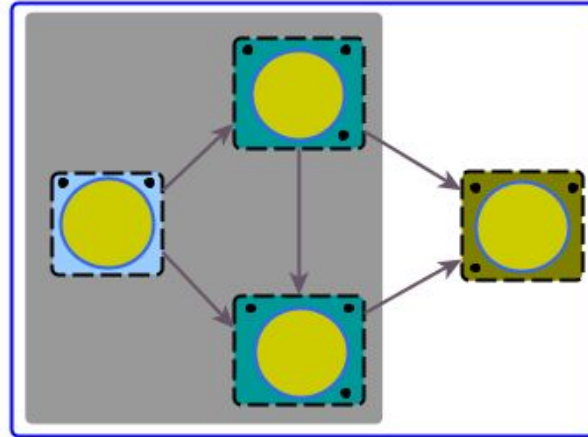
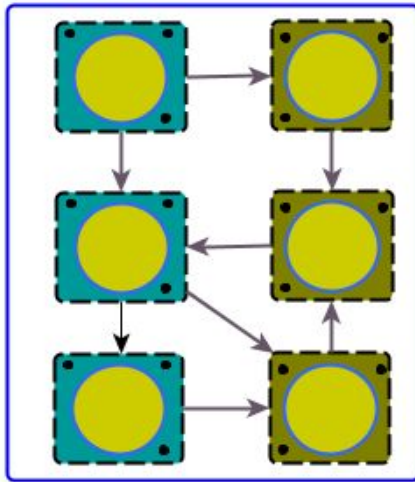
# Input/Output cardinality

- The input and output channel cardinality of a single building block or of a composition of building blocks can be computed considering the rules shown aside (the symbol “\*” denotes a cardinality *not statically defined*)
- A given building block may have different cardinality on the basis of the number of edge-nodes composing it and on the basis of the sequential node used for implementing edge-nodes



# Graphs that cannot be built with Building Blocks

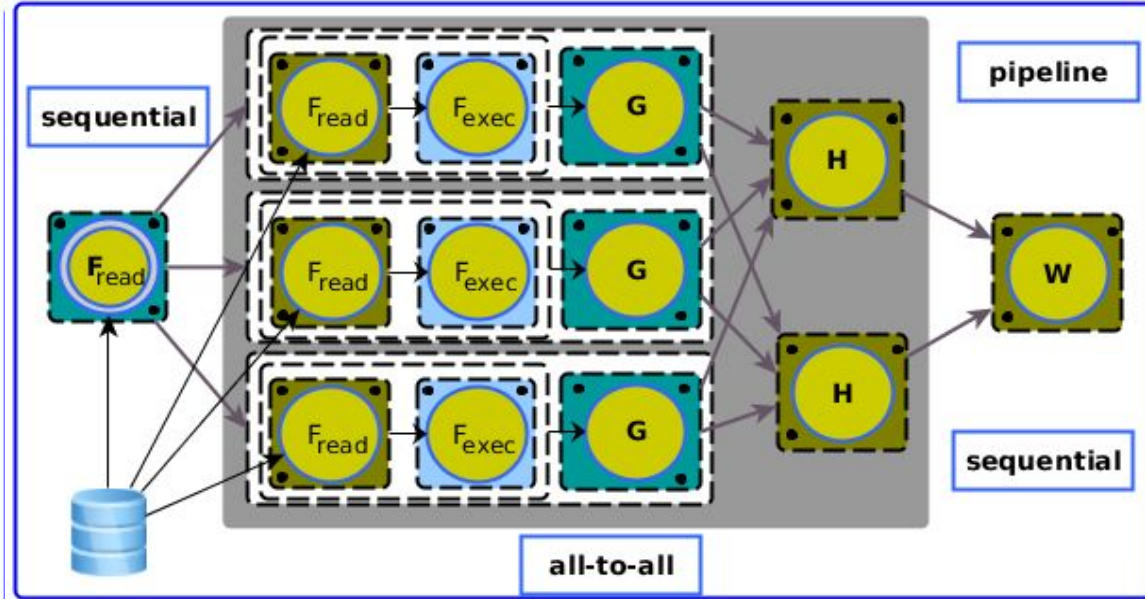
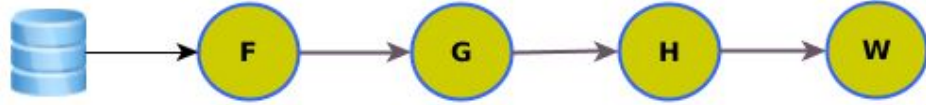
Examples of graphs that cannot be built with building blocks



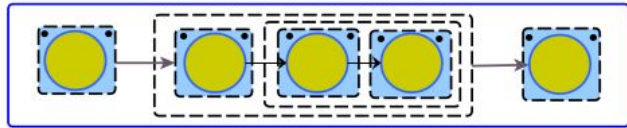
*Only a subset of all possible concurrency graphs can be built ....*

**... all the interesting ones (structured approach)**

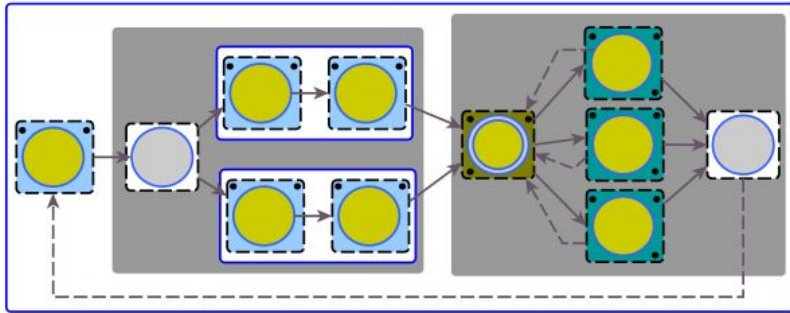
# Building Blocks composition example



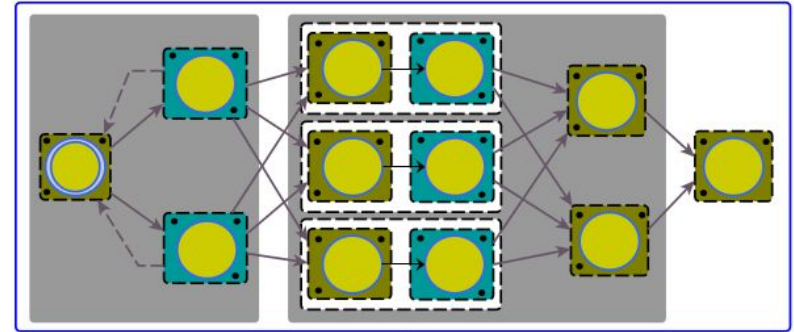
# Building Blocks composition examples



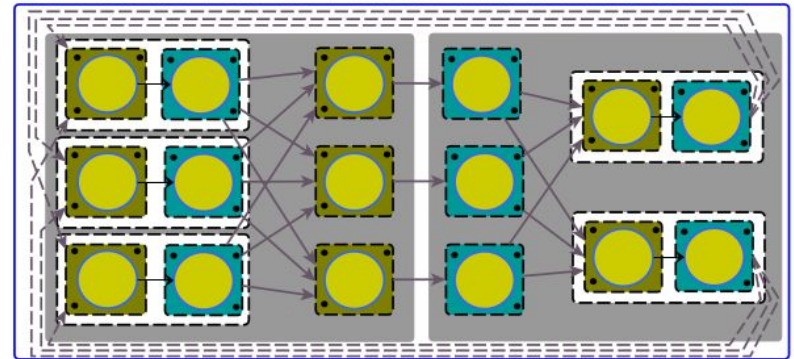
a)



b)



c)



d)

# Simple usage examples

```
#include <ff/ff.hpp>
using namespace ff;
// user-defined ...
Emitter E; // ... emitter
Collector C; // ... collector
Worker W;
// creating the pool of workers
std::vector<ff_node*> V;
for(int i=0;i<nworkers;++i)
    V.push_back(new Worker(W));
ff_farm farm(V,E,C);
farm.set_scheduling_ondemand();
farm.cleanup_workers();
if (farm.run_and_wait_end()<0)
    error("running farm");
```

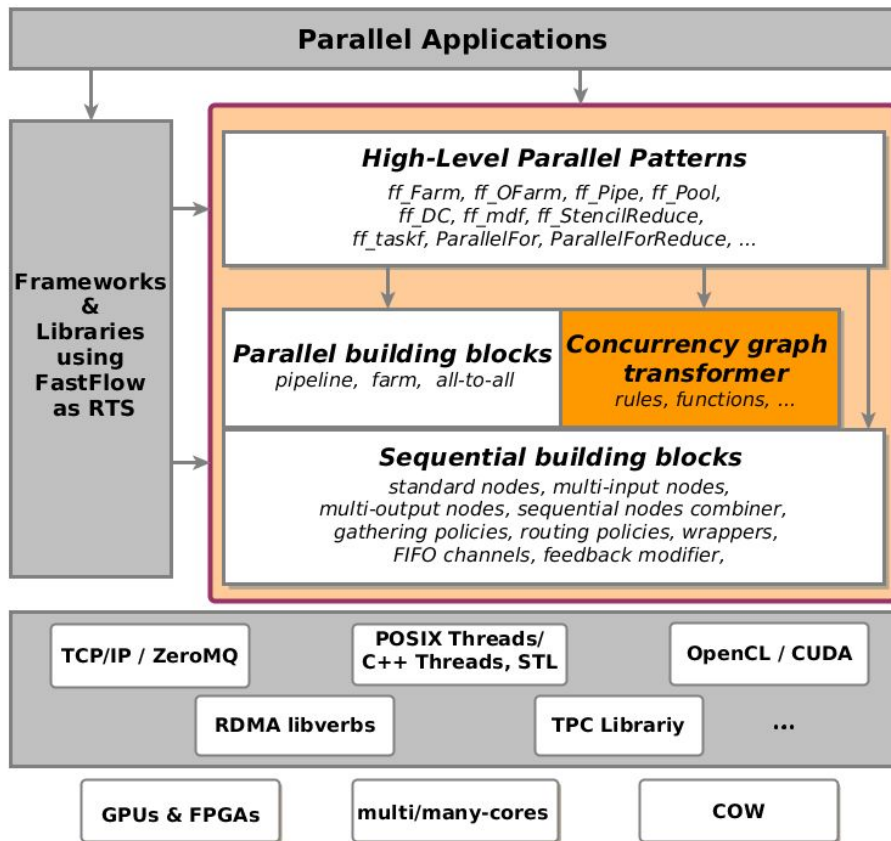
```
#include <ff/ff.hpp>
using namespace ff;
// user-defined workers
Worker1 W1; // standard node
Worker2 W2; // multi-output node
MultiInputHelper1 H1; // helper node
MultiInputHelper2 H2; // helper node

// creating the L-Workers
std::vector<ff_node*> V1;
for(int i=0;i<nworkers1;++i)
    V1.push_back(new ff_comb(H1,W1));

// creating the R-Workers
std::vector<ff_node*> V2;
for(int i=0;i<nworkers2;++i)
    V2.push_back(new ff_comb(H2,W2));

ff_a2a a2a;
a2a.add_firstset(V1, 0, true);
a2a.add_secondset(V2, true);
a2a.wrap_around();
if (a2a.run_and_wait_end()<0)
    error("running a2a");
```

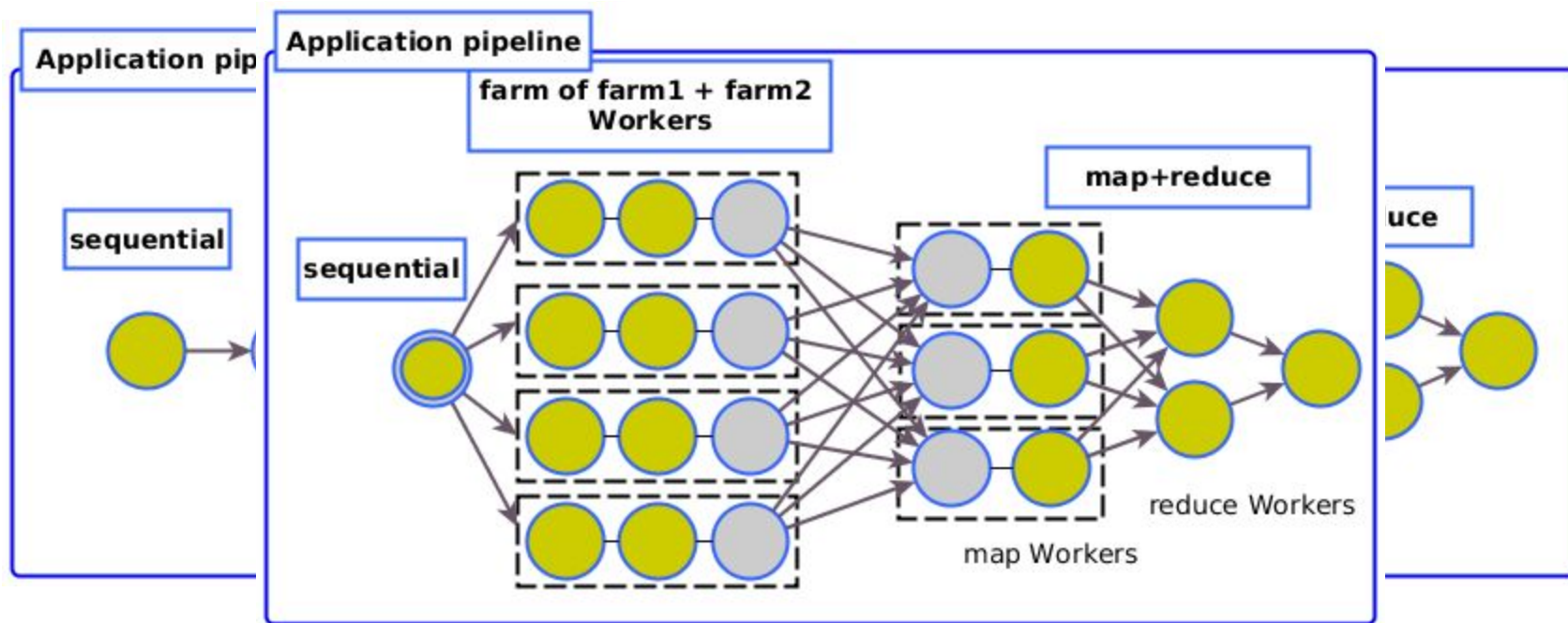
# Concurrency Graph Transformer



# Basic idea

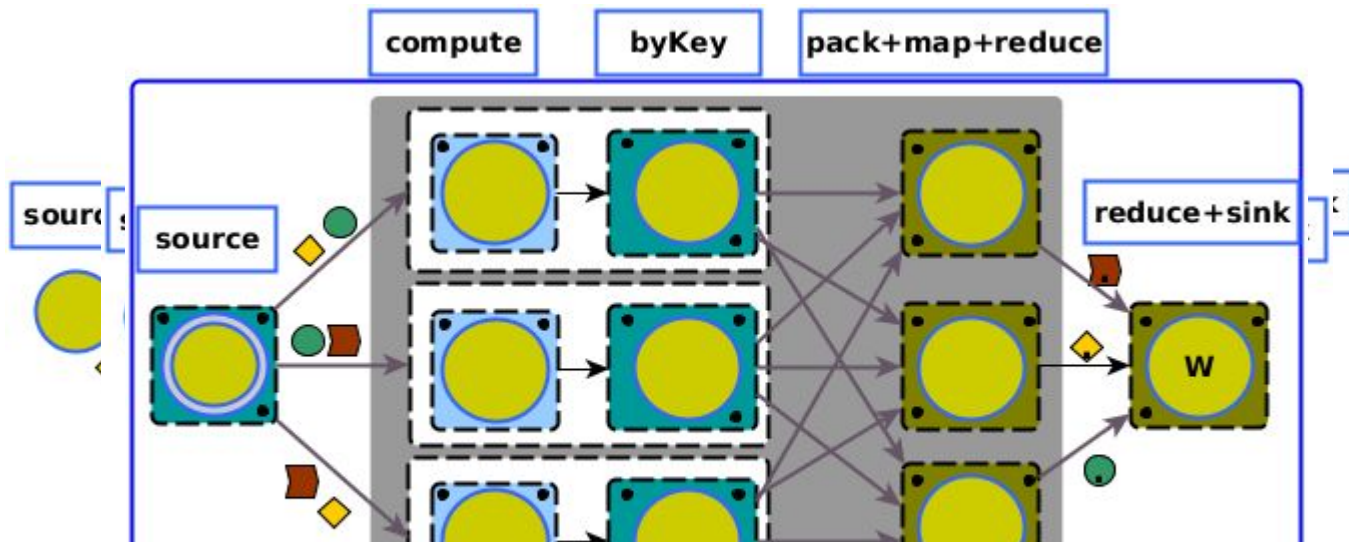
- The concurrency graph transformer provides functions to refactor the FastFlow graph
- **Objectives:** to reduce the number of nodes (i.e. default Emitters/Collectors) and to enable advanced combinations of BBs
- A simple interface function, called ***optimize\_static()***, is provided to the FastFlow programmer to automatically apply transformations to the pipeline composition of BBs
  - These transformations are applied only if specific conditions hold so to not modify the semantics of the transformed program (user-hints)
  - The programmer can force some transformations using a proper API provided by the layer taking the responsibility of the transformations
- Automatic transformations currently supported are:  
***nodes reduction, farm fusion, farm combine, All-to-All introduction***
- ***Extra-functional operations supported*** : *concurrency control* (blocking vs non-blocking), initial barrier removal, default-mapping of threads (enabled by default, can be disabled), ...

# Example



# Example

- Transformations that modify the original functional semantics might still be acceptable if acknowledged by the programmer



How to help the programmer to introduce such transformations in the FF's high-level layer? Specialized nodes?

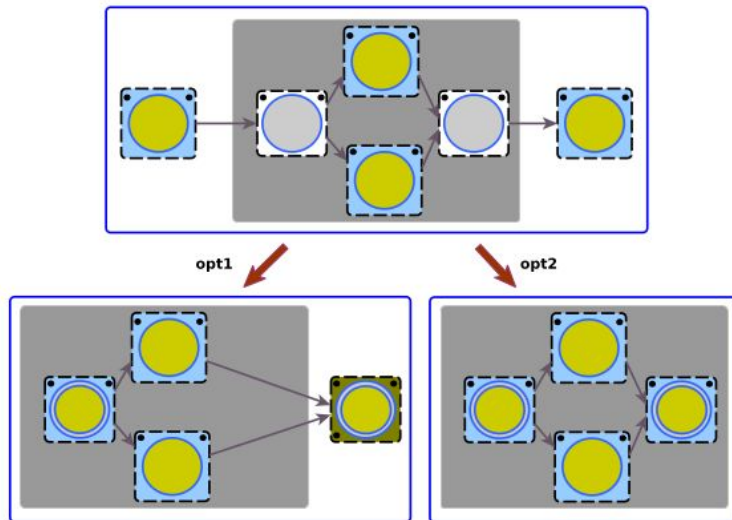
# Nodes reduction

- The ff\_Farm pattern uses by default an Emitter and Collector nodes (not user-defined)
- We may want to avoid having dedicated nodes performing only scheduling and gathering functionalities
- We can merge them with the previous/subsequent pipeline node
- Example: three-staged pipeline where the middle stage is a farm with default emitter/collector

```
... // creating Stage1, myFarm and Stage2
ff_Pipe<> pipe(Stage1, myFarm, Stage2);

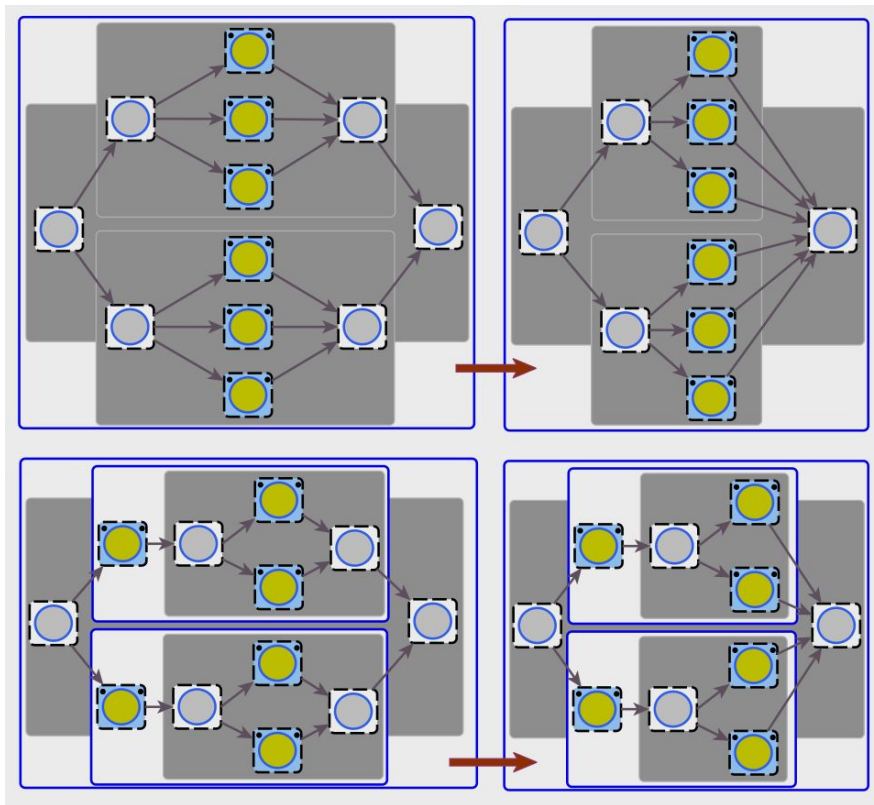
OptLevel opt;
opt.remove_collector =true;
opt.merge_with_emitter=true;
optimize_static(pipe, opt);

.... // adding the transformed pipe to another pipe
ff_Pipe<> pipe2(Stage0, pipe, Stage3);
...
pipe2.run();
```



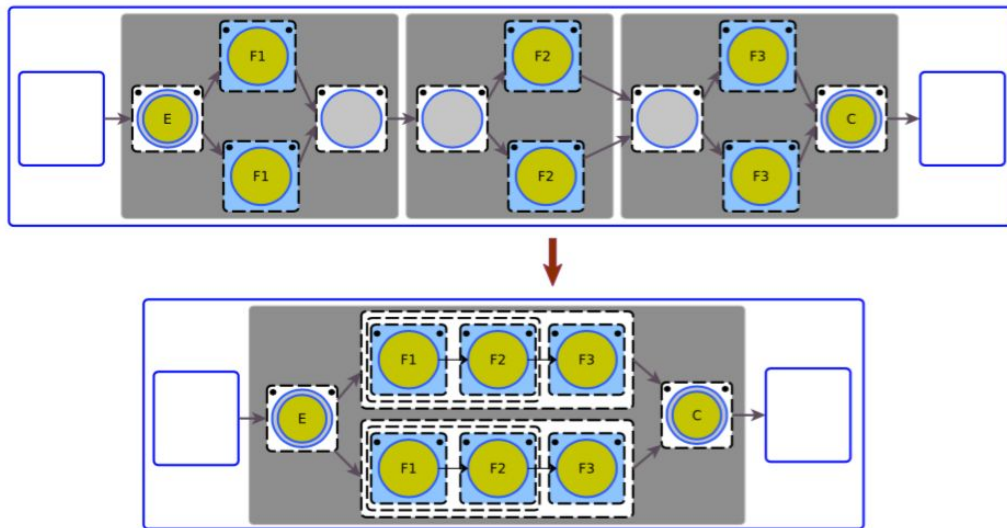
# Nodes reduction

- Extra nodes added when nesting a farm BB into a farm or a pipeline
  - Top: the results produced by the inner farm Workers will be directly forwarded to the collector of the outermost farm which will do the collection
  - Bottom: each Worker is a pipeline where the farm is the last stage containing a default collector.



# Farm fusion

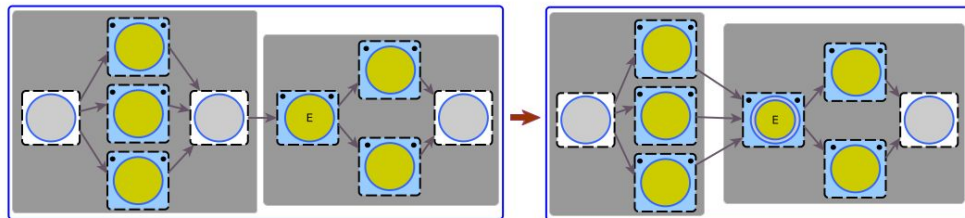
- Two farms are consecutive stages of a pipeline and they have the same parallelism degree (number of Workers)
- The result of the fusion is a **unique farm** merging the two original ones.
- Besides reducing nodes also in this case, this optimization may be useful
  - to reduce latency (number of hops)
  - to avoid bottlenecks between the Collector of the first farm and the Emitter of the second one



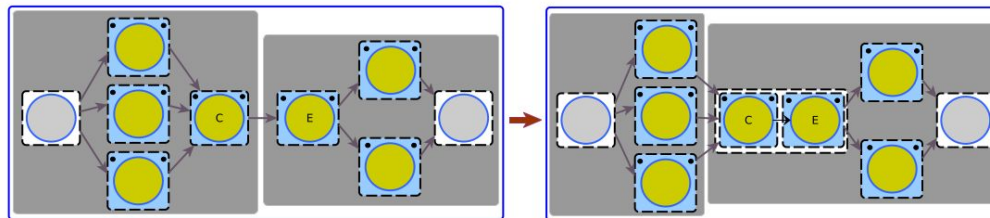
# Farm combine

- Two farms in pipeline without any constraint on their parallelism degree and on their emitter/collector nodes (they can be either default or custom nodes)
- Similar operation to “farm fusion” but limited to the Emitter and Collector nodes

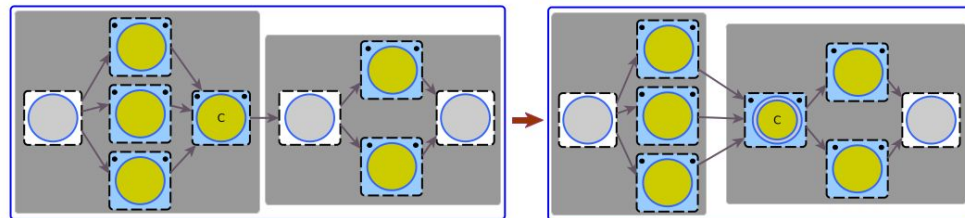
Case1



Case2



Case3

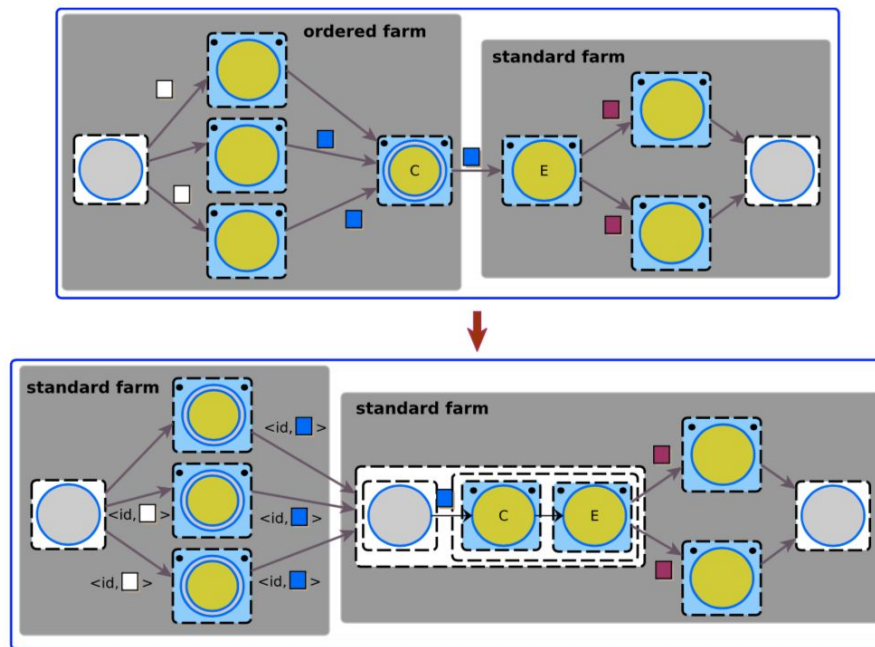


# Farm combine

- An interesting case of the farm combine transformation is given when one of the two farms to be combined is an ordered farm instance
- This case introduces some constraints in order to preserve the ordering of stream elements

- Case1: The first farm is a standard farm while the second one is an ordered farm. Simplest case (not shown)

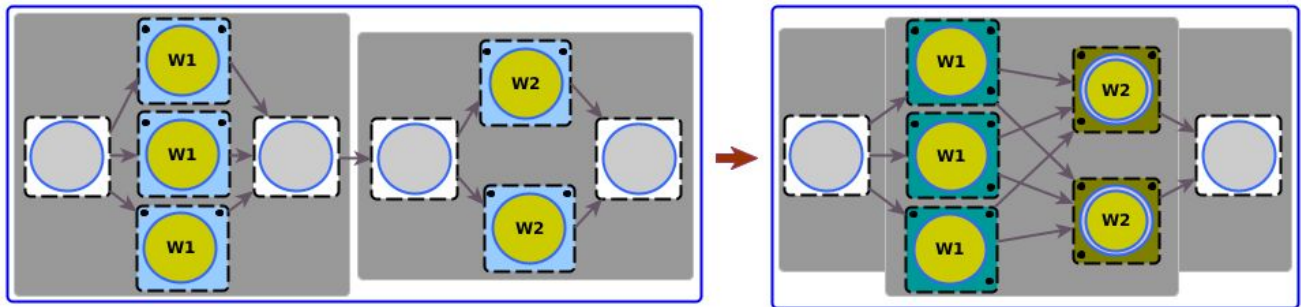
- Case2: The opposite of Case1. The new Emitter of the first farm adds unique identifiers to the input elements. The Emitter of the second farm combines a special ordering collector which orders and remove the identifiers to the results.



# All-to-all introduction

- Two farms in pipeline regardless their parallelism degree
- We want to remove the potential bottleneck introduced by the Collector and Emitter without changing the amount of concurrency of the worker nodes

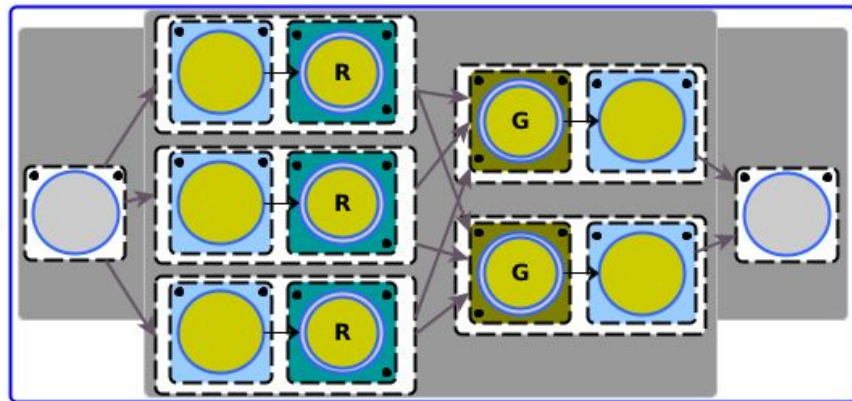
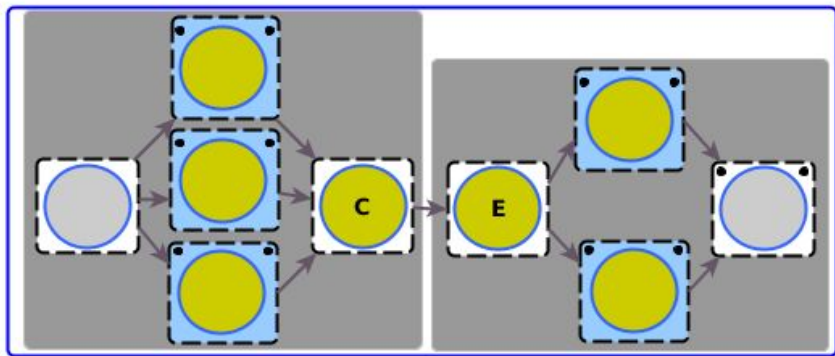
## Example



- The Workers of the first and second farm are **automatically transformed into multi-output and multi-input nodes**, respectively
- The Workers of the first farm are added to the L-Worker set of the all-to-all while the Workers of the second farm are added to the R-Worker set of the same all-to-all building block

# All-to-all introduction

- What about if the Emitter and/or the Collector are user-defined?
- The transformation has to be “guided” by the user

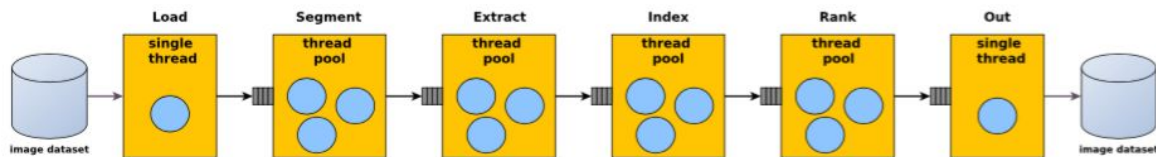


- In the figure: **R=E** and **G=C**.
- R and G can be any sequential composition of nodes such that they are a multi-output node and a multi-input node, respectively

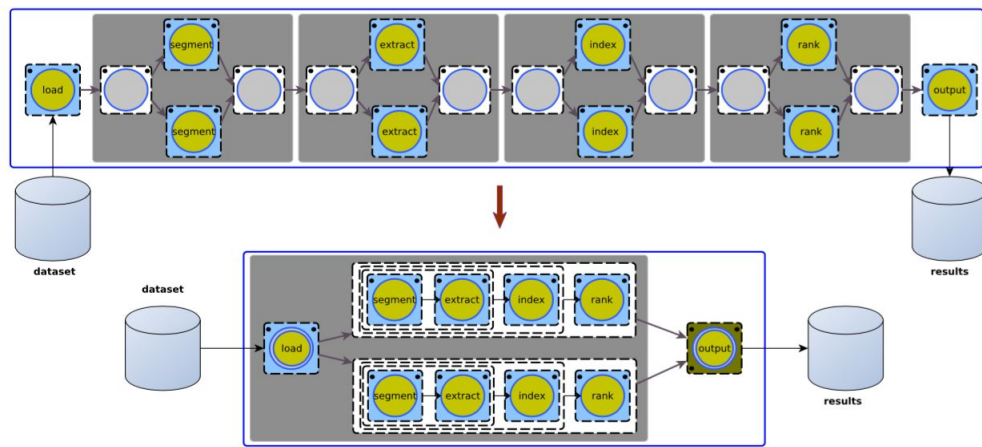
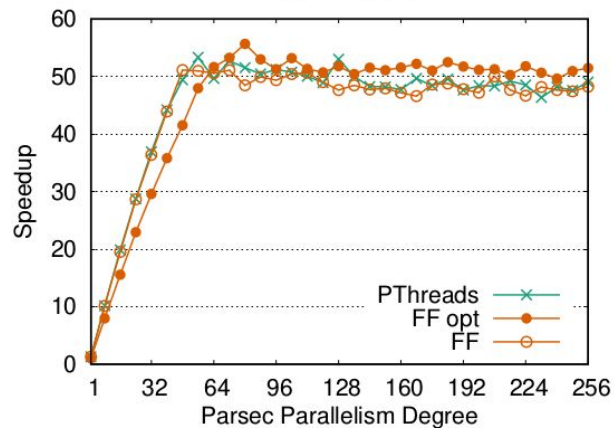
```
template<typename R_t, typename G_t>
const ff_pipeline combine_farms(ff_farm& farm1, const R_t *R,
                              ff_farm& farm2, const G_t *G,
                              bool merge);
```

# Ferret Example

- Ferret applications from the PARSEC benchmark suite (Princeton Application Repository for Shared-Memory Computers)
- Ferret: content-based similarity search of feature-rich data such as audio, images, video, and 3D shapes



Ferret, KNL platform



# Ferret FastFlow code

```
struct Load: ff_node_t<long,load_data> { // first stage
    load_data *svc(long*) { <business logic code> };
} In;
struct Segment:ff_node_t<load_data,seg_data> { // second stage
    seg_data *svc(load_data *in) { <business-logic code> };
};
struct Extract:ff_node_t<seg_data,extr_data> { // third stage
    extr_data *svc(seg_data *in) { <business-logic code> };
};
struct Index:ff_node_t<extr_data,vec_query_data> { // fourth stage
    vec_query_data *svc(extr_data *in) { <business-logic code> };
};
struct Rank:ff_node_t<vec_query_data,rank_data> { // fifth stage
    rank_data *svc(vec_query_data *in) { <business-logic code>};
};
struct Output:ff_node_t<rank_data> { // sixth stage
    void *svc(rank_data *in) { <busines-logic code> };
} Out;
auto farm1 = ...; auto farm2 = ...; // creating farms stages
auto farm3 = ...; auto farm4 = ...;
ff_Pipe<> pipe(In, farm1, farm2, farm3, farm4, Out);
OptLevel opt;
opt.remove_collector =true;
opt.merge_with_emitter=true;
opt.merge_farms      =true;
optimize_static(pipe, opt);
pipe.run_and_wait_end(); // pipeline execution
```

# Ferret transformations

```
auto farm1 = ...; auto farm2 = ...; // creating farms stages  
auto farm3 = ...; auto farm4 = ...;  
ff_Pipe<> pipe(In, farm1, farm2, farm3, farm4, Out);
```

```
OptLevel opt;
```

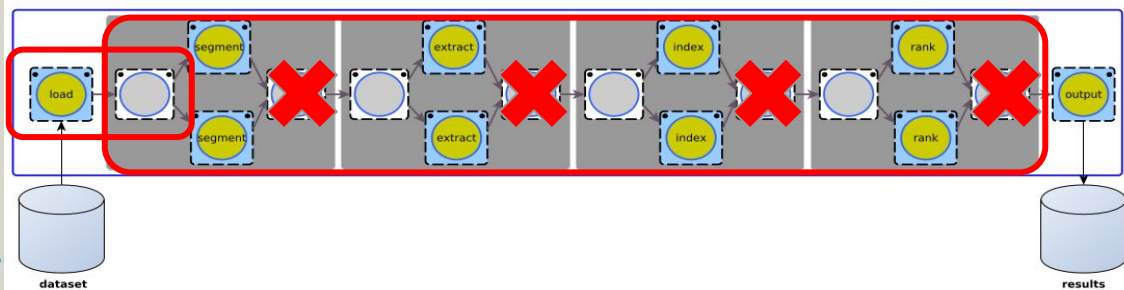
```
opt.remove_collector=true;
```

```
opt.merge_with_emitter=true;
```

```
opt.merge_farms=true;
```

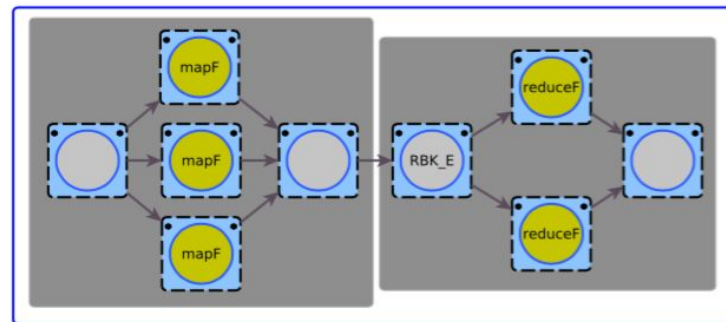
```
optimize_static(pipe, opt);
```

```
pipe.run_and_wait_end(); // pipelin
```

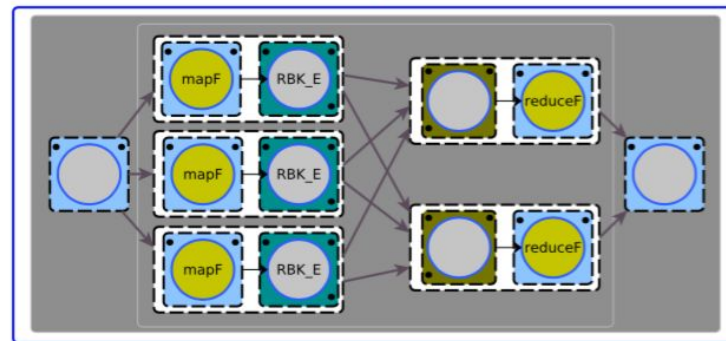


# All-to-all introduction example

- To evaluate the advantage of A2A introduction, we consider a classical benchmark used to evaluate framework for big data analytics: WordCount (PiCO library)
- A pipeline of two parallel transformations (map-reduce)
  - a “flatmap” operator in charge of producing for each word of a text file a pair (1, word)
  - a “reduce” operator merging all the pairs with the same word in a unique pair (n, word), where n is the number of pairs with that word as key
- Shuffling can be applied to avoid bottlenecks

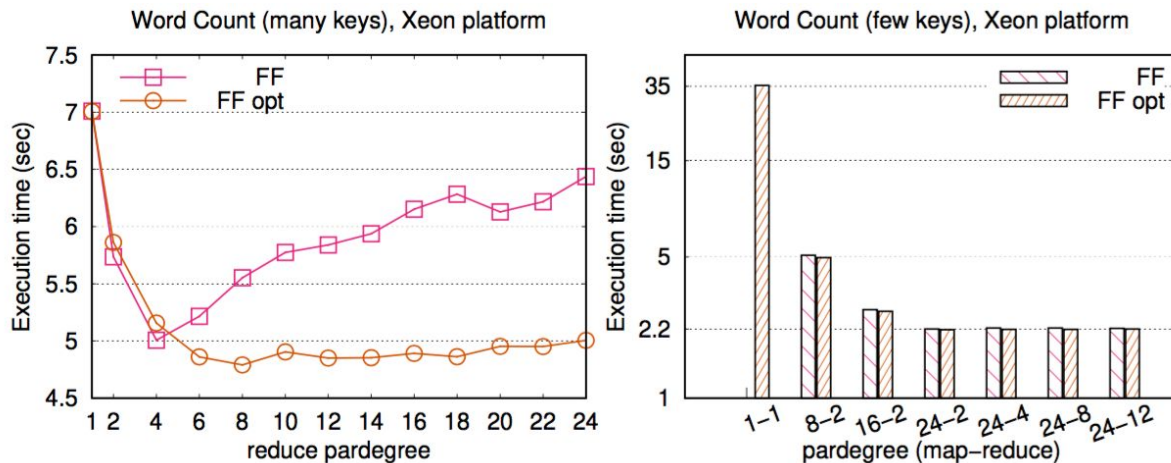


↓  
**Map/FlatMap + ReduceByKey  
transformation using the  
all-to-all building block**



# All-to-all introduction example

- Results obtained using the PiCO framework implemented on top of FastFlow. WordCount test on a Intel Xeon platform 24-cores 28-threads, with different number of keys that change the cost of the distribution
- More than 350K different words (keys) in the “many keys” scenario, only 10K in the “few keys” case



- Parallelism degree of the FlatMap fixed to 24. Slightly better performance and more stable behavior by increasing the degree of parallelism in the reduce phase
- In the test with few keys, the reduce phase is fixed to use parallelism degree equal to 2. No additional cost of using the A2A here

# Further research on BBs (some options)

## 1. Introduce more advanced static transformations

Some transformations need “*helper nodes*” to fulfill the Composition Rules. **These nodes could be introduced automatically** so to enlarge the options

## 2. Transformations applied at run-time while the program is running (e.g., the initial program can be compiled into multiple versions):

- To adapt to fluctuating workloads (in addition to concurrency throttling)
- To use them in different places of the program (as for the *ParallelFor*)