Introduction to FastFlow programming

Massimo Torquati
<massimo.torquati@unipi.it>
SPM lecture 1

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

Lessons calendar

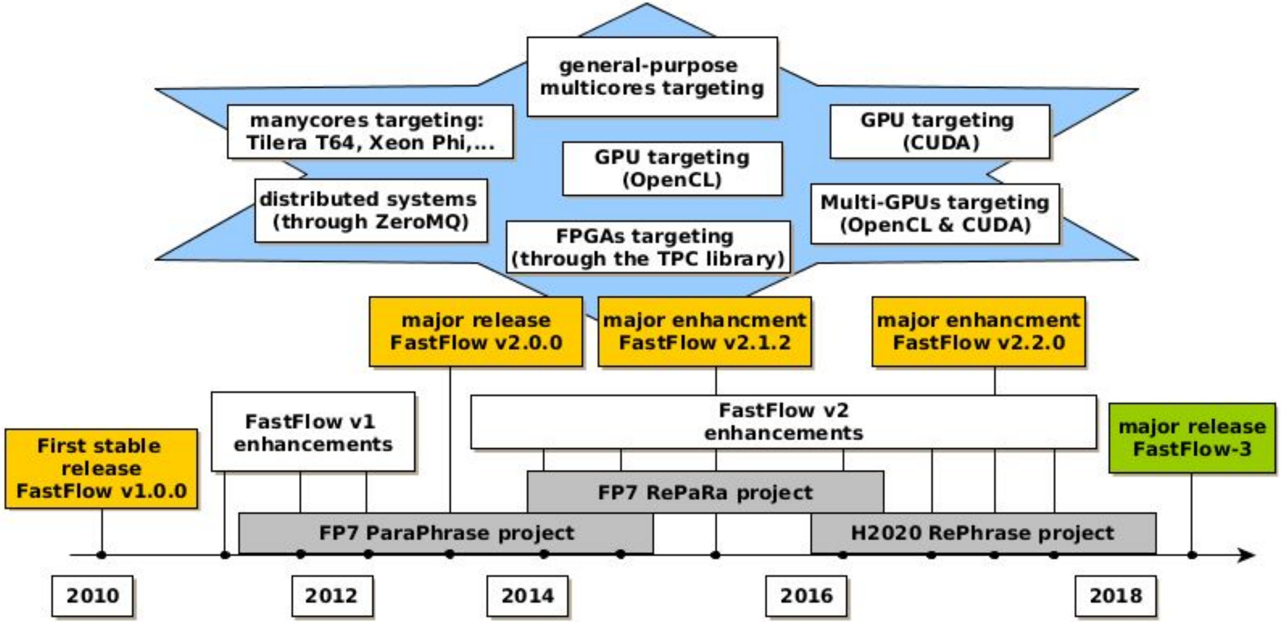
	When	What	Notes
Lesson1	Monday, April 12th	Introduction to FF, seq & pipeline	today!
Lesson2	Thursday, April 15th	Farm skeletons	
Lesson3	Monday, April 19th	Map, ParallelFor	
Lesson4	Thursday, April 22nd	Farm/ParallelFor examples, MDF, D&C patterns	
Lesson5	Monday, April 26th	FastFlow Building Blocks	**9-10.30am** (with s smaller break)
Lesson6	Monday, May 5th	WindFlow DSL example, Concurrency throttling in the ff_farm	
Lesson7	Monday, May 10th	Application examples (PARSEC, FastFilesCompressor)	
Lesson8	Monday, May 17th	Threads mapping, Memory allocators	**To be confirmed!**

What is FastFlow

- Research project started in 2010 (UniPI + UniTo)
- It is a parallel programming library written in C++ promoting structured parallel programming both at the RTS level as well as at the higher level
    - RTS level by using tiny components called ***building blocks***
    - Application level by using well-known ***parallel patterns***
- It aims to tackle the *"3Ps" challenge* for multi/many-core platforms: *<u>P</u>rogrammability, <u>P</u>erformance, <u>P</u>ortability*
- Project actively maintained and utilized in several EU funded research initiatives
- Currently, FastFlow is moving to distributed systems
    - There exists an old proof-of-concept implementation of the FastFlow library for Clusters of Workstations based on ZeroMQ (not maintained anymore)
    - *<u>Opportunities for master thesis on this topic!</u>*

History

- About a decade of research
- 100+ research papers
- Several Ph.D. thesis
- Some IT industries tested/used FastFlow



Download

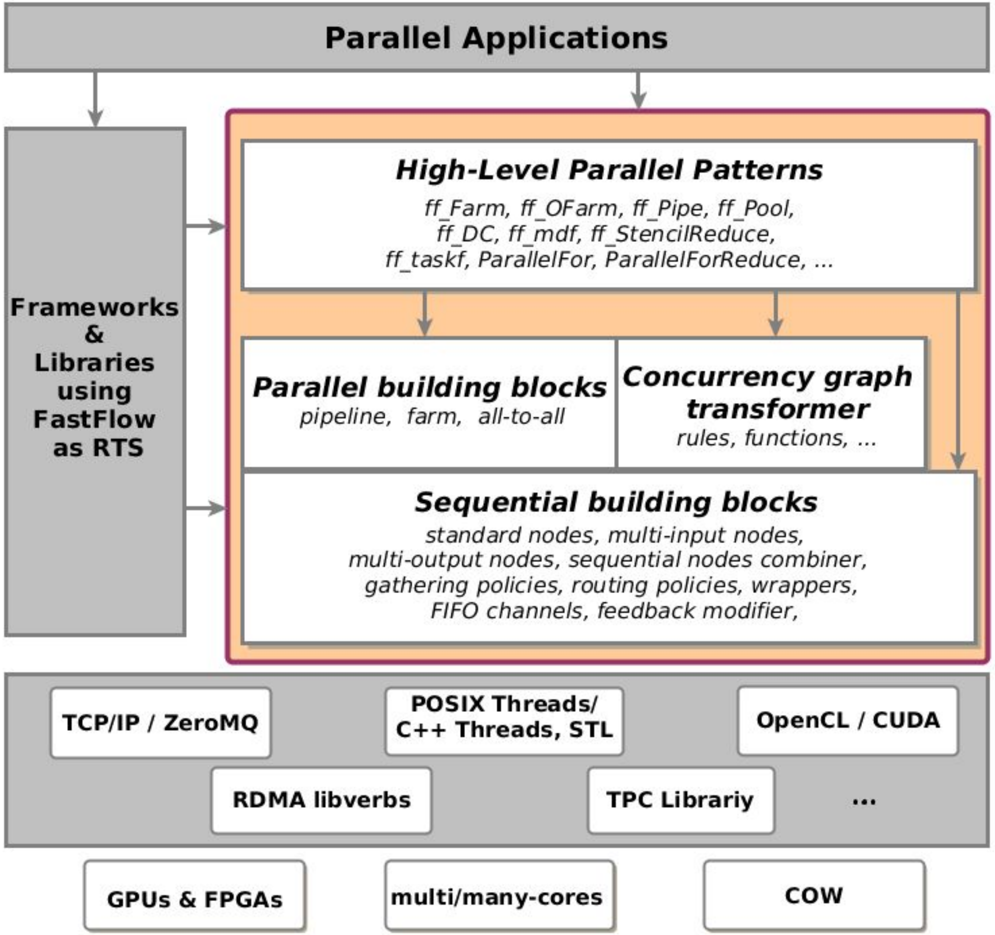
- The latest version (v3) available on GitHub (previously on Sourceforge):
  - git clone https://github.com/fastflow/fastflow.git it creates a fastflow directory
  - git pull inside the directory to get the latest update
  - **NOTE:** remember to run the Bash script *mapping_string.sh*
- It is possible to download the previous version (v2.2.0) containing simple examples and a tutorial (tab Releases on github)
- Notes:
  - From v2 to v3 the interface changed (minor modifications, mainly for the ff_farm)
  - To compile you need a recent C++ compiler
  - Compile with **-std=c++17** flag to avoid warnings
  - FastFlow is a class library (i.e., only header files) therefore do not forget to use the -O3 flag
  - You have to always include <*ff/ff.hpp*> first and then the others include needed (eg. <ff/ParallelFor.hpp>)

FastFlow stack

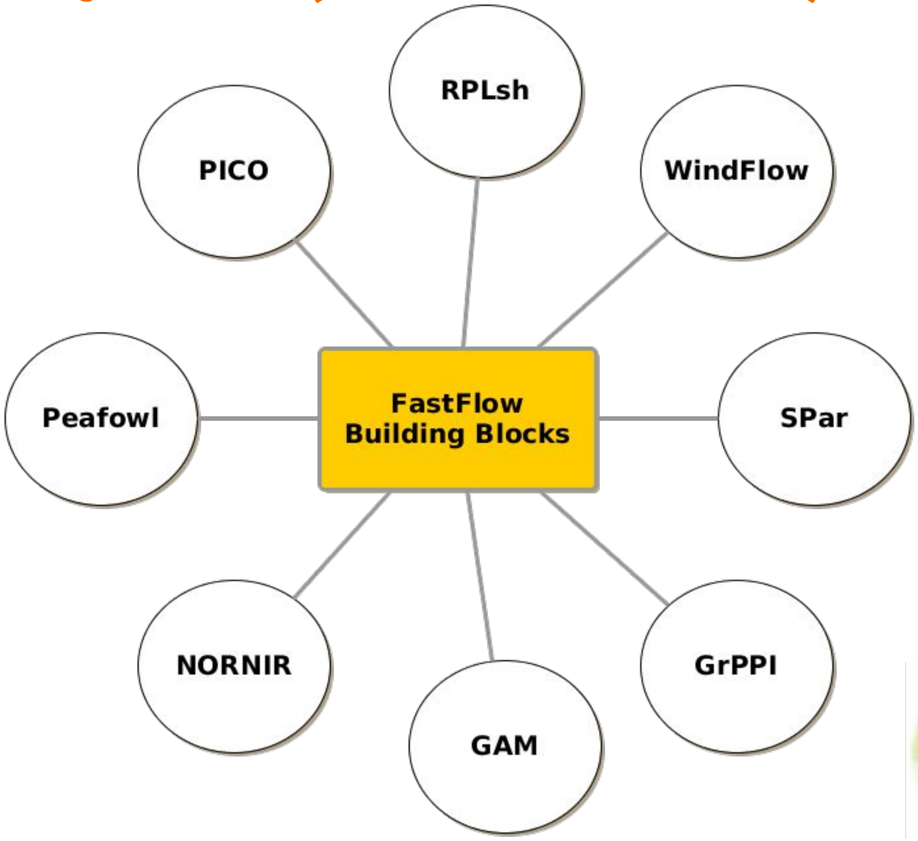
- Header-only library
- Multi-level APIs
- High-level Parallel Patterns targeting the application programmer
- Parallel and Sequential Building Blocks targeting the RTS programmer

Open-source LGPLv3

Home: http://calvados.di.unipi.it/fastflow
GitHub: http://github.com/fastflow/fastflow



FastFlow ecosystem (research tools)



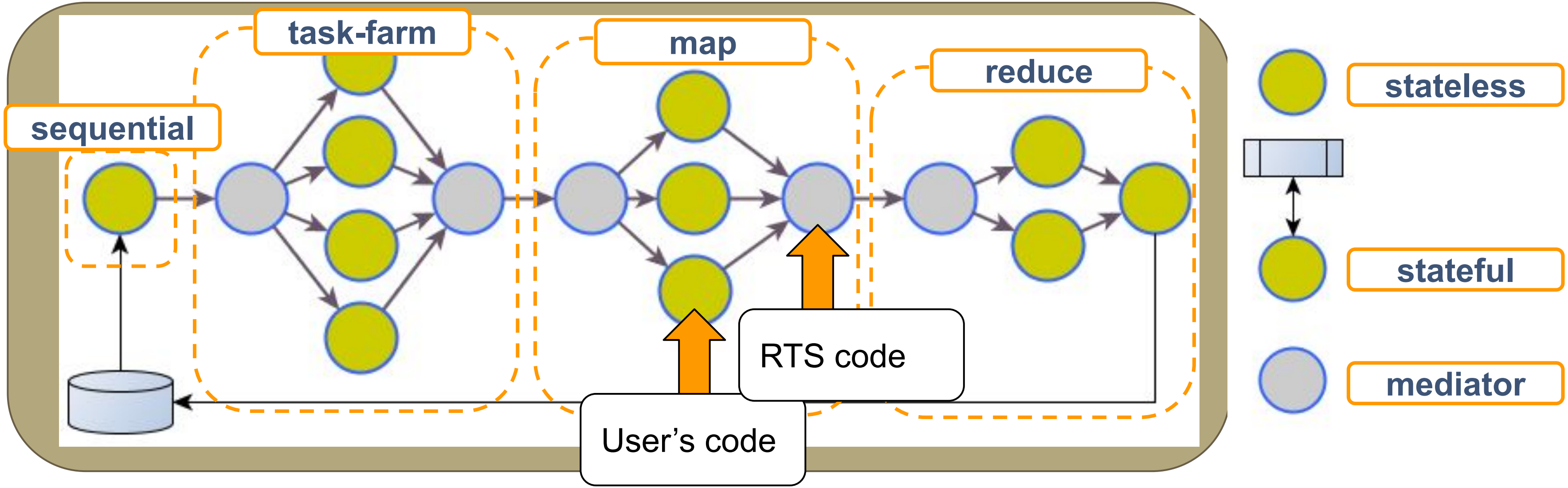
PARAPHRASE

REPARA
Reengineering and Enabling Performance and poweR of Applications

REPHRASE

FastFlow programming model (basic concepts)

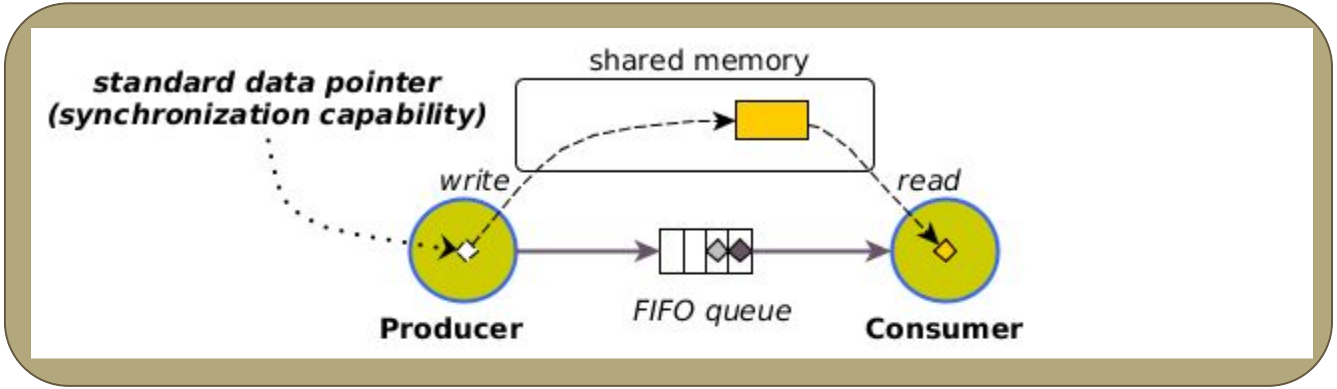
- An application is a ***directed graph*** whose **nodes** are computing entities (either sequential or parallel) and **edges** are channels carrying ***data references (pointers)***



- Nodes can be ***stateless*** or ***stateful***. In the most common case, a sequential node is a thread.

FastFlow programming model (basic concepts)

- Nodes synchronize through ***message-passing*** for accessing shared data
- Idea: "*Don't communicate by sharing memory; share memory by communicating*"
    - This the Go language motto (https://golang.org/doc/codewalk/sharemem/)



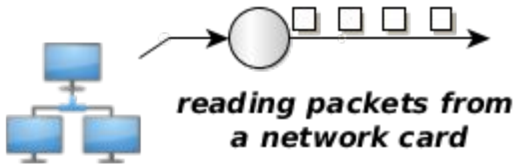
- Writing a pointer into the FIFO queue enforces write ordering also on non TSO (Total Store Ordering) memory consistency models

FastFlow programming model (stream concept)

- ***Data Stream*** *is a first class concept in the FastFlow library*
- A stream is a sequence of data values (possibly infinite), coming from one or more sources. The items of a stream coming from the same source typically have the same data type.
- Streams can be either primitive ("*eso-stream*"), i.e. coming from a real source, for example a sensor, a network interface, a file, etc., or can be generated directly by the application ("*endo-streams*"), mainly by unrolling the iterations of a loop
- Terminology: stream *source/sink*, the first/last stage of a FastFlow pipeline

FastFlow programming model (stream concept)

- Streams coming from a real data source



  - Data arrive with a given input rate
    - there could be fluctuations in the input rate, bursts, data losses, etc.
  - The system should guarantee:
    - to sustain the maximum input rate
    - to process the single stream item within a given service time

FastFlow programming model (stream concept)

- Endo-stream: data stream produced by unrolling loops
  - The data source is a software module
  - Typically the length is bounded by an upper-bound
  - Typically generated at **full speed**, the most common objective is to reduce the overall completion time

```
for(i=start; i<stop; i+=step)
   allocate data for an item/task
   create a task
   send out the task to the next filter
```

FastFlow communication channels

- A communication channel connecting two FastFlow nodes is implemented by using a non-blocking SPSC FIFO queue (Single-Producer Single-Consumer First-In First-Out)
- The classical concurrency control approach for concurrent data structures is lock-based, which usually implies passive waiting when the queue is full or empty (**blocking** concurrency control protocol)
- SPSC queues are interesting because they can be implemented in a very efficient way on CMP platforms without the need of locks nor atomic instructions (i.e., CAS, LL/SC)
  - Lamport's circular buffer under Sequential Consistency memory model
- **Non-blocking** protocols use active-waiting loops with some back-off policies before trying again if the operation fails (e.g., push/pop operations in a concurrent queue)
  - **Obstruction-free progress**: a thread executed in isolation for a finite number of steps is capable to make progress and to terminate.
  - **Lock-free progress:** a failure/suspension of a thread does not prevent progress of other threads (there could be starvation, though). A lock-free algorithm is obstruction-free.
  - **Wait-free progress** (the strongest progress property): all concurrent operations in the algorithm are guaranteed to always complete in a finite number of steps. A wait-free algorithm is lock-free.

FastFlow communication channels

- FastFlow channels may have either **bounded** or **unbounded** capacity. They are implemented by the uSWSR_Ptr_Buffer class (file ff/ubuffer.hpp) and we will refer to them as **uSPSC** queues
- The implementation of the uSPSC queue is based on the bounded non-blocking **SPSC** queue (file ff/buffer.hpp -- class SWSR_Ptr_Buffer)
  - SPSC is basically a Lamport's buffer working also for Weak Memory Consistency Models
- FastFlow users do not have to deal with FastFlow channels!

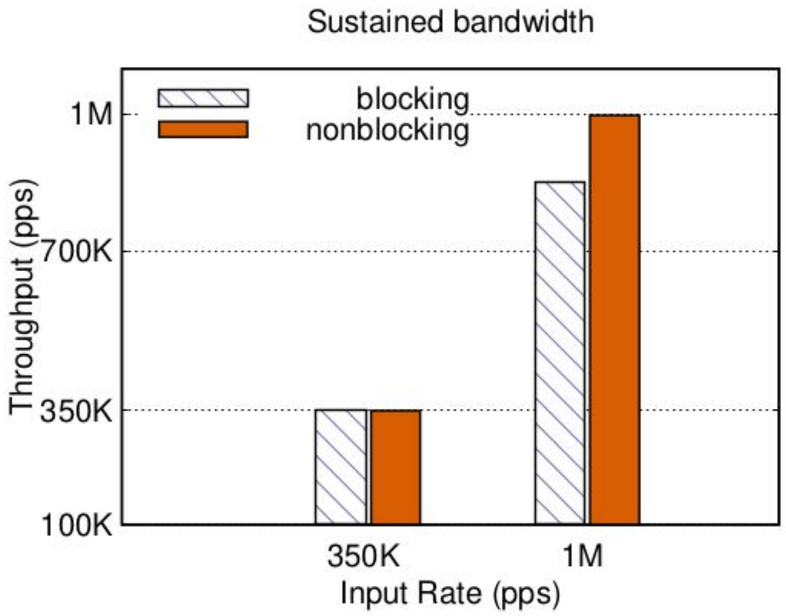
SPSC non-blocking FIFO buffer pseudo-code (uSPSC queues are a bit more complex)

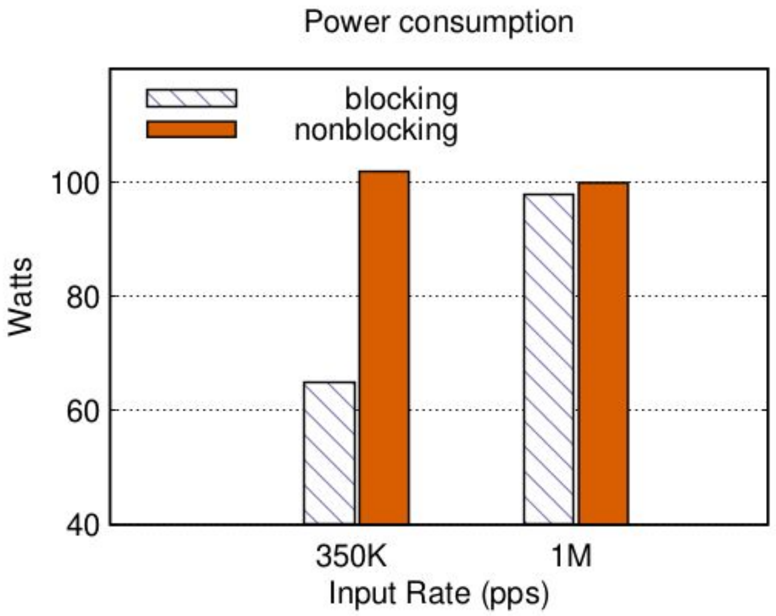
```
bool push(void* data) {
  if (buf[pwrite]==NULL) {
    WMB(); // write-memory-barrier
    buf[pwrite] = data;
    pwrite+=(pwrite+1>=size)?(1-size):1;
    return true;
  }
  return false;
}
```

```
bool pop(void** data) {
  if (buf[pread]==NULL)
    return false;
  *data = buf[pread];
  buf[pread]=NULL;
  pread+=(pread+1>=size)?(1-size):1;
  return true;
}
```

FastFlow communication channels

- Performance vs Power Consumption (uSPSC)





- The blocking version protects push/pop operations with standard mutexes/CVs, and if the queue is empty the consumer will block

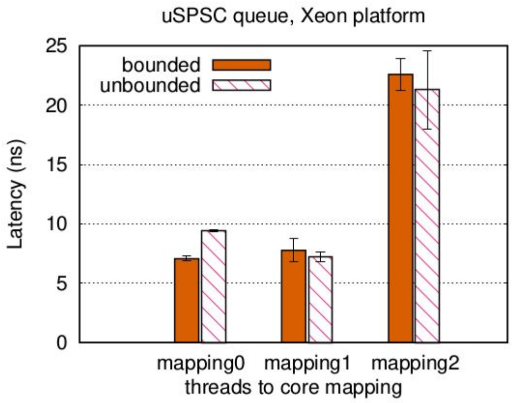
FastFlow communication channels

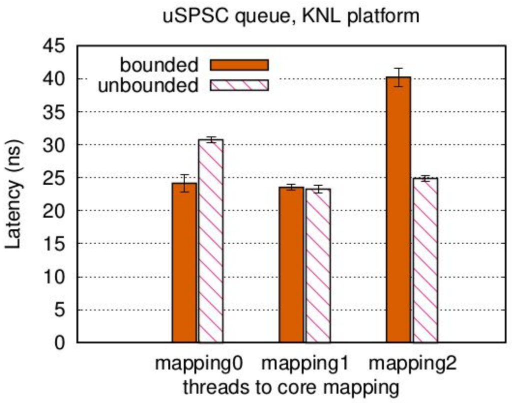
Xeon and KNL:

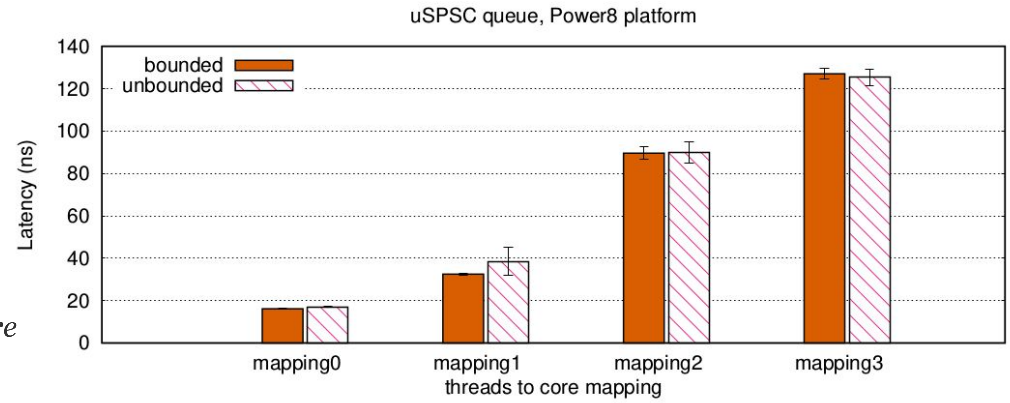
- mapping0: same core 2 contexts
- mapping1: 2 cores same CPU
- mapping2: 2 CPUs

Power:

- mapping0: same core 2 contexts
- mapping1 : 2 cores same CMP
- mapping2: same CPU 2 CMPs
- Mapping3: 2 CPUs







Details of the implementation of the uSPSC in:
M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "*An Efficient Unbounded Lock-Free Queue for Multi-core Systems*" Euro-Par 2012, doi:10.1007/978-3-642-32820-6_65

High-Level Parallel Patterns

- **Stream parallel**: ff_Pipe, ff_Farm, ff_OFarm
- **Data parallel**: ff_Map, ParallelFor*, poolEvolution, ff_stencilReuced*
- **Task parallel**: ff_DC, ff_mdf

They try to address application programmers' needs

FastFlow Name	Pattern description
`ff_Pipe`	Pipeline pattern modeling a data-flow *sequence* of sequential and parallel patterns. It allows to implements both linear and non-linear compositions of parallel patterns. The FastFlow implementation of this pattern is based on the *pipeline* building block.
`ff_Farm`, `ff_OFarm`	Task-farm pattern modeling functional replication. The `ff_OFarm` pattern guarantees to preserve input ordering. Both are built on top of the *farm* building block.
`ff_Map`	In this pattern a function F is applied to all elements of the input collection. The parallel calculation of the function F over disjoint partitions of the initial collection does not require any communication/synchronization except for a barrier at the end. It is implemented on top of the *farm* building block by using the `broadcast` and `gather_all` distribution and collection policies. The FastFlow library provides also an implementation based on the `ParallelForReduce` pattern.
`ff_mdf`	It models the Macro Data-Flow execution model. A program is interpreted by focusing on the functional dependencies among data [20, 69].
`poolEvolution`	This pattern models the evolution of a population according to the principles typical of evolutionary computing [12].
`ff_DC`	This pattern models *Divide & Conquer* computations [109].
`ParallelFor`, `ParallelForReduce`	They model data-parallel computations, allowing the parallelization of loops with independent iterations and also having reduction variables [119].
`ff_stencilReduceOCL`, `ff_stencilReduceCUDA`	These patterns model iterative stencil plus reduce computations targeting GPU accelerators either using OpenCL or using CUDA code [23, 8].

FastFlow sequential node (ff_node_t class)

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

- **svc_init()** is called once at the beginning each time the thread associated to the node is (re-)started
- **svc()** is called for each input item present in one of the input queues. Its return value can be:
  - A memory pointer to the result data
  - ***EOS*** (End-Of-Stream)
  - ***GO_ON***
  - A few other special values can be returned
- **svc_end()** is called before terminating the node (or when the node is put to sleep).
  NOTE: It is not possible to send any “final” result into the output queues.

```
class myClass: public ff_node_t<IN_t, OUT_t>{
public:
   OUT_t* svc(IN_t * in) {
       <business logic code of the node
         producing the output task (out)>;
       return out;
  }
   int svc_init() { .......; return 0; }
   void svc_end() { ....... }

 protected/private:
 <local-state, if any>
};
```

Sequential node life-cycle

- **svc_init()** is called once at the beginning each time the thread associated to the node is (re-)started
- **svc()** is called for each input item present in one of the input queues. Its return value can be:
  - A memory pointer to the result data
  - *EOS* (End-Of-Stream)
  - *GO_ON*
  - A few other special values can be returned
- **svc_end()** is called before terminating the node (or when the node is put to sleep).
  NOTE: It is not possible to send any "final" result into the output queues.
- **eosnotify()** is called each time an EOS message is received by the node. It is possible to send data out into the output queue(s).

Sequential node life- cycle (simplified):

```
do {
  if (svc_init() < 0) break;
  do {
    in = input_channel.pop();
    if (in == EOS) {
      // if this method has been redefined, the user's method
      // is called and informed that the EOS has arrived
      eosnotify();
      output_channel.push(EOS);
    } else {
      out = svc(in);  // it calls the business logic code
      if (out == GO_ON) continue;
      output_channel.push(out);
    }
  } while(out != EOS);
  svc_end();
} while(true);
```

Standalone node

- This is just an example to understand the life-cycle of a node
  - The FastFlow graph is composed by a single node

- The output is:
  - Hello. I'm going to start
  - Hi! (0)

    Hi! (1)

    …

    Hi! (5)
  - Goodbye!
- The method run_and_wait_end is needed to call the base class methods run() and wait() to start the thread associated to the node and to wait for the termination of the node, respectively.

```cpp
#include <ff/ff.hpp>
using namespace ff;
struct myNode:ff_node_t<int> {
  int svc_init() {
    std::cout << "Hello. I'm going to start\n";
    counter = 0;
    return 0;
  }
  int* svc(int*) {
    if (++counter > 5) return EOS;
    std::cout << "Hi! (" << counter << ")\n";
    return GO_ON; // keep calling the svc method
  }
  void svc_end() { std::cout << "Goodbye!\n"; }
  // starts the node and waits for its termination
  int  run_and_wait_end(bool=false) {
    if (run() < 0) return -1;
    return wait();
  }
  long counter;
};
int main() {
  myNode mynode;
  return mynode.run_and_wait_end();
}
```

Source and Sink nodes

Source node creating a stream of Tasks

```
struct Source: ff_node_t<Task> {
  Task *svc(Task *) {
     // generates N tasks and then EOS
     for(long i=0;i<N; ++i) {
        ff_send_out(new Task);
      }
     return EOS;
  };
};
```

Sink node absorbing the stream

```
struct Sink: ff_node_t<Task> {
  Task *svc(Task * task) {
     // do something with the task
     do_Work(task);
     delete task;
     return GO_ON; // it does not send out task
  };
};
```

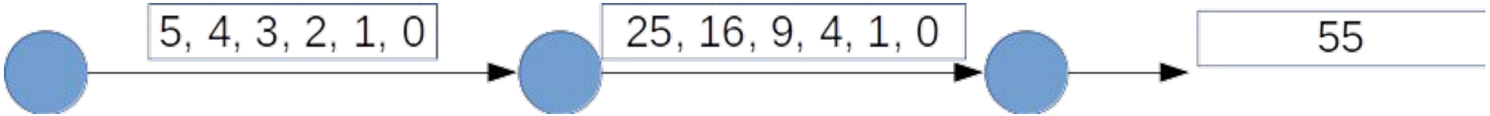
Pipeline

```
struct Source: ff_node_t<myTask> {
  myTask *svc(myTask *) {
       for(long i=0;i<10;++i)
          ff_send_out(new myTask(i));
       return EOS;
}};
struct Stage: ff_node_t<myTask> {
  myTask *svc(myTask *task) {
       return task;
}};
struct Sink: ff_node_t<myTask> {
  myTask *svc(myTask* task) {
       f3(task);
       return GO_ON;
}};
Source _1;
Stage  _2;
Sink  _3;
ff_Pipe<> pipe(_1,_2,_3);
pipe.run_and_wait_end();
```

- Pipeline stages are nodes (ff_node_t)
- A pipeline itself is a node
  - It is possible to build pipe of pipes
- A node, for each input, may generate zero, one, or many outputs
- The first stage is the *source* node
  - It generates 10 items and then the **EOS**
- The last stage is the *sink* node that executes function ‘f3’ and then **GO_ON**
- In this example, the middle stage is just a forwarder node

Pipeline example

- Computing the sum of the square of the first N numbers by using a pipeline




```
// 3-stage pipeline
ff_Pipe<> pipe( first, second, third );
pipe.run_and_wait_end();
```

```
// 1st stage
struct firstStage: ff_node_t<float> {
  firstStage(const size_t len):len(len) {}
  float* svc(float *) {
    for(long i=0;i<len;++i)
      ff_send_out(new float(i));
    return EOS; // End-Of-Stream
  }
  const size_t len;
};
```

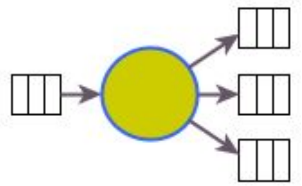
Possible extention: think about how to avoid using many new/delete

```
// 2nd stage
struct secondStage: ff_node_t<float> {
  float* svc(float *task ) {
      float &t = *task;
      t = t*t;
      return task;
  }
};
```

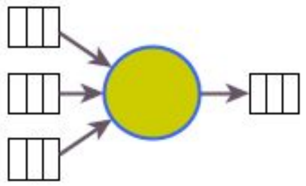
```
// 3rd stage
struct thirdStage: ff_node_t<float> {
  float* svc(float *task ) {
      float &t = *task;
      sum +=t;
      delete task;
      return GO_ON;
  }
  void svc_end() { std::cout << "sum = " << sum << "\n"; }
  float sum = {0.0};
};
```

Multi-Input and Multi-Output nodes

- Single-Input Single-Output node
- ff_node_t<IN_t, OUT_t>
- ff_send_out(data)



- Single-Input Multiple-Output node
- **ff_monode_t**<IN_t, OUT_t>
- ff_send_out_to(data, channel-id) channel-id goes from 0 to N-1



- Multiple-Input Single-Output node
- **ff_minode_t**<IN_t, OUT_t>
- ff_get_channel_id() to know from which channel we received the input (from 0 to N-1)
- *fromInput*() return true if the input comes from one of the input channels (useful in case of inputs coming both from input and feedback channels)