

# Introduction to FastFlow programming

**SPM lecture, November 2015**

Massimo Torquati <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy



# Objectives

- Have a good idea of the FastFlow framework
  - how it works and its main features
  - also, weakness and strength points
- To be able to write simple (but not-trivial) FastFlow programs

# What is FastFlow

- FastFlow is a parallel programming framework written in C/C++ promoting pattern based parallel programming
- It is a joint research work between Computer Science Department of University of Pisa and Torino
- It aims to be usable, efficient and flexible enough for programming heterogeneous multi/many-cores platforms
  - multi-core + GPGPUs + Xeon PHI + FPGA .....
- FastFlow has also a distributed run-time for targeting cluster of workstations

# Downloading and installing FastFlow

- Supports for Linux, Mac OS, Windows (Visual Studio)
  - The most stable version is the Linux one
    - we are going to use the Linux (x86\_64) version in this course
- To get the latest svn version from Sourceforge

```
svn co https://svn.code.sf.net/p/mc-fastflow/code/ fastflow
```

- creates a fastflow dir with everything inside (tests, examples, tutorial, ....)
- To get the latest updates just cd into the fastflow main dir and type:

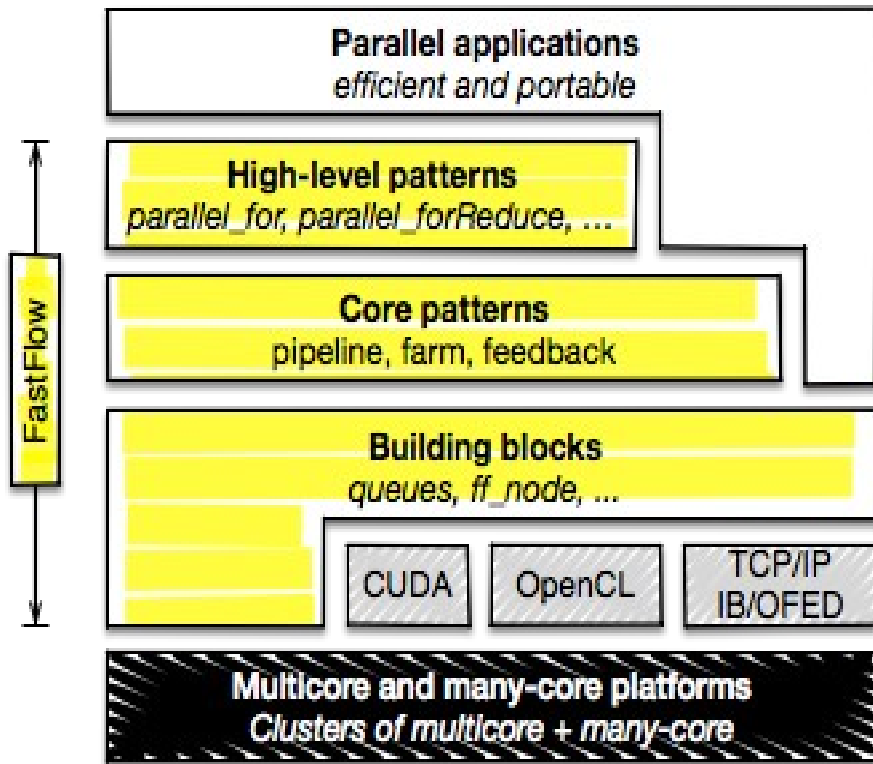
```
svn update
```

- The run-time (i.e. all you need for compiling your programs) is in the *ff* directory (i.e. *fastflow/ff*)
  - NOTE: FastFlow is a class library not a plain library
- You need: make, g++ (with C++11 support, i.e. version  $\geq 4.7$ )

# The FastFlow tutorial

- The FastFlow tutorial is available as pdf file on the FastFlow home page under “Tutorial”
  - <http://mc-fastflow.sourceforge.net> (aka [calvados.di.unipi.it](http://calvados.di.unipi.it) )
  - “FastFlow tutorial” (“PDF File”)
- All tests and examples described in the tutorial are available as a separate tarball file: **fftutorial\_source\_code.tgz**
  - can be downloaded from the FastFlow home (“Tests and examples – source code tarball”)
- In the tutorial source code there are a number of very simple examples covering almost all aspects of using pipeline, farm, ParallelFor, map, mdf.
  - Many features of the FastFlow framework are not covered in the tutorial yet
- There are also a number of small (“more complex“) applications, for example: image filtering, block-based matrix multiplication, mandelbrot set computation, dot-product, etc...

# The FastFlow layers



- C++ class library
- Promotes (high-level) structured parallel programming
- It aims to be flexible and efficient enough to target **multi-core, many-core** and **distributed heterogeneous systems**.
- Layered design:
  - **Building blocks** minimal set of mechanisms: channels, code wrappers, combinators.
  - **Core patterns** streaming patterns (*pipeline* and *task-farm*) plus the *feedback* pattern modifier
  - **High-level patterns** aim to provide flexible reusable parametric patterns for solving specific parallel problems

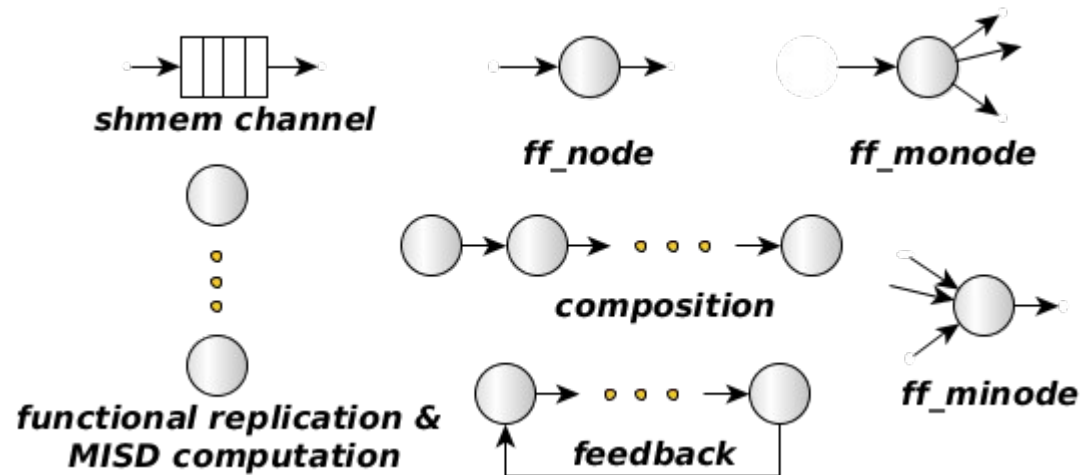
<http://mc-fastflow.sourceforge.net>

<http://calvados.di.unipi.it/fastflow>

# What FastFlow provides

- FastFlow provides **patterns** and **skeletons**
  - Pattern and algorithmic skeleton represent the same concept but at different abstraction level
- Stream-based parallel patterns (pipe, farm) plus a pattern modifier (feedback)
- Data-parallel patterns (map, stencil-reduce)
- Task-parallel pattern (async function execution, macro-data-flow)
- FastFlow does not provide implicit memory management of data structures
  - In almost all patterns memory management is left to the user
  - Memory management is a very critical point for performance

# Building blocks



- Minimal set of efficient mechanisms and functionalities
- Nodes are concurrent entities (i.e. POSIX threads)
- Arrows are channels implemented as SPSC lock-free queue
  - bounded or unbounded in size

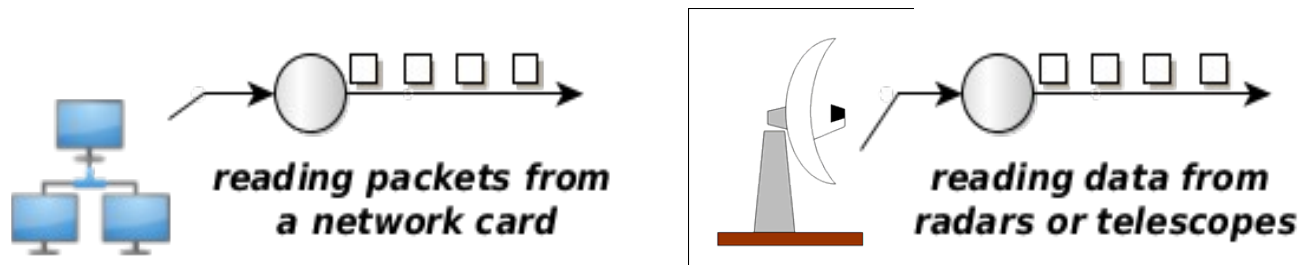


# Stream concept (recap)

- Sequence of values (possibly infinite), coming from a source, having the same data type
  - Stream of images, stream of network packets, stream of matrices, stream of files, .....
- A streaming application can be seen as a work-flow *graph* whose nodes are computing nodes (sequential or parallel) and arcs are channels bringing streams of data.
- Streams may be either “*primitive*“ (i.e. coming from HW sensors, network interfaces, .....) or can be generated internally by the application (“*fake stream*”)
- Typically in a stream based computation the first stage receives (or reads) data from a source and produces tasks for next stages.

# Real and Fake streams

- “*real streams*“

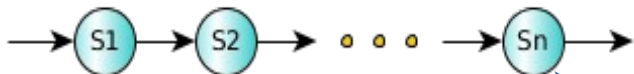


- In these cases it is really important to satisfy minimum processing requirements (bandwidth, latency, etc...) in order to not lose data coming from the source
- “*fake streams*“: streams produced by unrolling loops
  - You don't have an “infinite“ source of data
  - The source is a software module
  - Typically less stringent constraints

```
for(i=start; i<stop; i+=step)
  allocate data for a task
  create a task
  send out the task
```

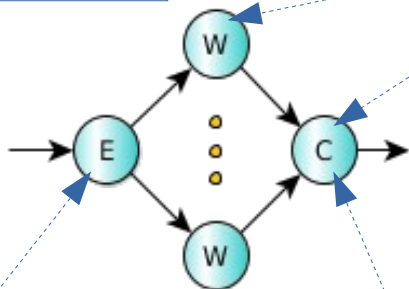
# Stream Parallel Patterns in FastFlow

pipeline



```
ff_Pipe<myTask> pipe(S1,S2,...,Sn);  
pipe.run_and_wait_end();
```

task-farm



ff\_node

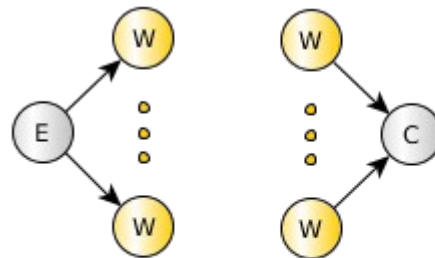
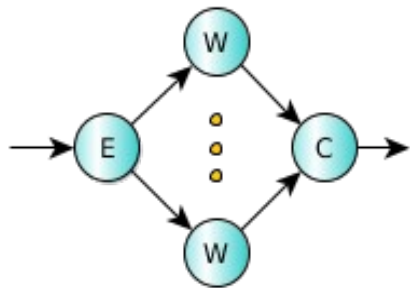
```
std::vector<std::unique_ptr<ff_node> > Warray;  
ff_Farm<myTask> farm(std::move(Warray),E, C);  
farm.run_and_wait_end();
```

Emitter:  
schedules input data items

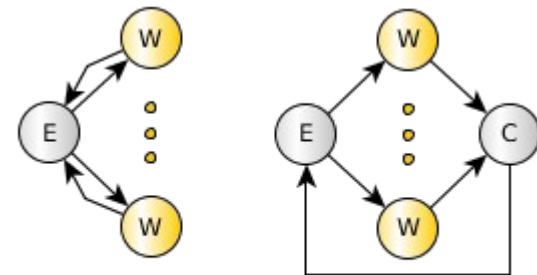
Collector:  
gathers results

# Stream Parallel Patterns (“core” patterns)

task-farm

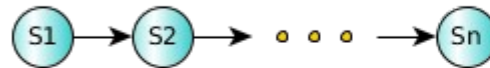
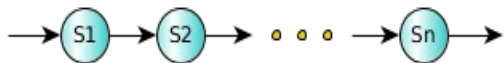


*task-farm*

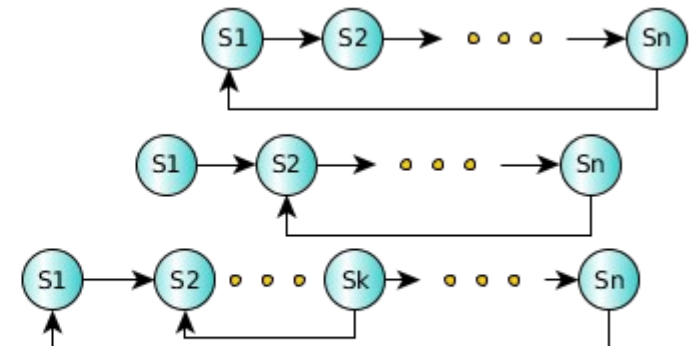


*task-farm + feedback*

pipeline



*pipeline*

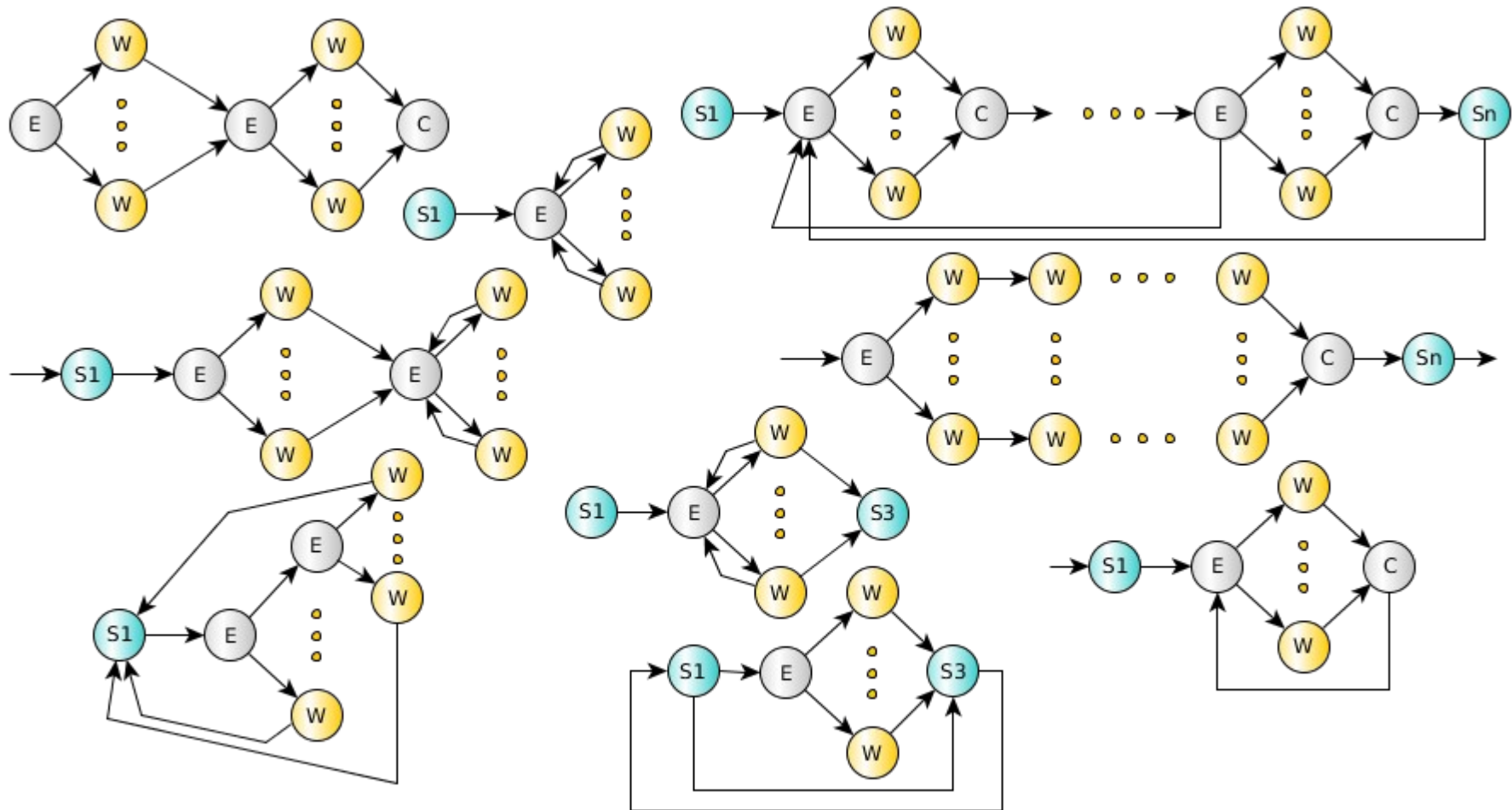


*pipeline + feedback*

Specializations

Patterns

# Core patterns composition



**pipeline + task-farm + feedback**

# Core patterns: sequential *ff\_node*

## code wrapper pattern

```
struct myNode: ff_node_t<TIN,TOUT> {  
  int svc_init() { // optional  
    // called once for initialization purposes  
    return 0; // <0 means error  
  }  
  TOUT *svc(TIN * task) {  
    // do something on the input task  
    // called each time a task is available  
    return task; // also EOS, GO_ON, ....  
  };  
  void svc_end() {  
    // called once for termination purposes  
    // called if EOS is either received in input  
    // or it is generated by the node  
  }  
};
```

- A sequential *ff\_node* is an active object (thread)
- Input/Output tasks (stream elements) are memory pointers
- The user is responsible for memory allocation/deallocation of tasks
  - FF provides a memory allocator (not introduced here)
- Special return values:
  - *EOS* means End-Of-Stream
  - *GO\_ON* means “I have no more tasks to send out, give me another input task (if any)”

# ff\_node: generating and absorbing tasks

## code wrapper pattern

```
struct myNode1: 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;  
    };  
};
```

```
struct myNode2: 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  
    };  
};
```

- Typically myNode1 is the first stage of a pipeline, it produces tasks by using the *ff\_send\_out* method or simply returning task from the svc method
- Typically myNode2 is the last stage of a pipeline computation, it gets in input tasks without producing any outputs

# Core patterns: *ff\_pipe*

## pipeline pattern

```
struct myNode1: ff_node_t<myTask> {
    myTask *svc(myTask *) {
        for(long i=0;i<10;++i)
            ff_send_out(new myTask(i));
        return EOS;
    }
};
struct myNode2: ff_node_t<myTask> {
    myTask *svc(myTask *task) {
        return task;
    }
};
struct myNode3: ff_node_t<myTask> {
    myTask *svc(myTask* task) {
        f3(task);
        return GO_ON;
    }
};
myNode1 _1;
myNode2 _2;
myNode3 _3;
ff_Pipe<> pipe(_1,_2,_3);
pipe.run_and_wait_end();
```

- *pipeline* stages are *ff\_node(s)*
- A *pipeline* itself is an *ff\_node*
  - It is easy to build pipe of pipe
- **ff\_send\_out** can be used to generate a stream of tasks
- Here, the first stage generates 10 tasks and then EOS
- The second stage just produces in output the received task
- Finally, the third stage applies the function f3 to each stream element and does not return any tasks

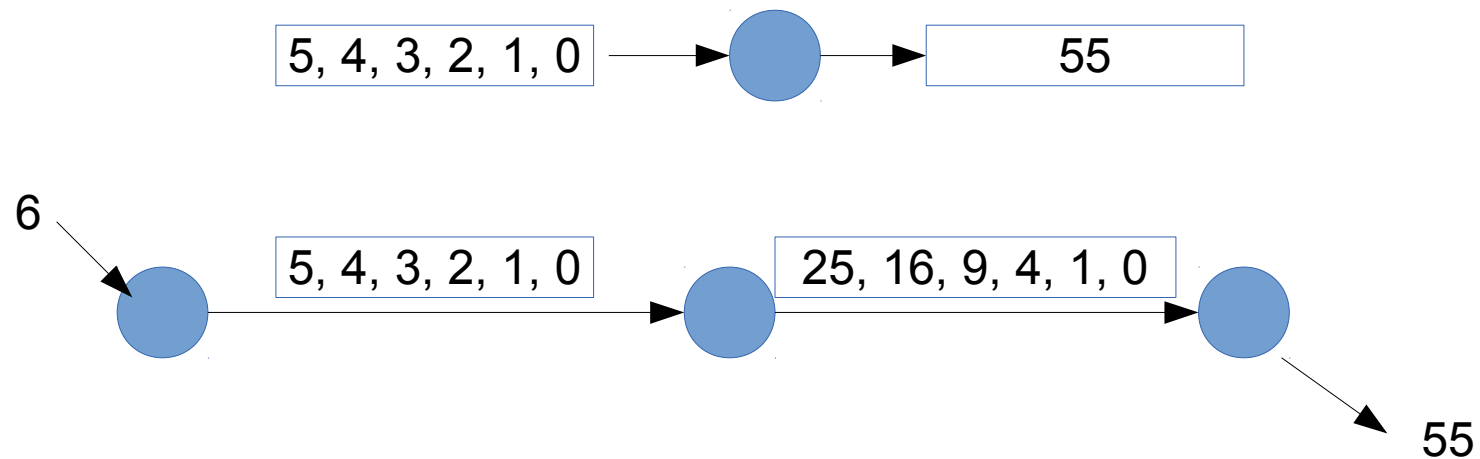


# Simple *ff\_Pipe* examples

- Let's take a look at 2 simple tests in the FastFlow tutorial:
  - `hello_pipe.cpp`
  - `hello_pipe2.cpp`
- How to compile:
  - Suppose we define the env var `FF_HOME` as (bash shell):
    - `export FF_HOME=$HOME/fastflow`
  - `g++ -std=c++11 -Wall -O3 -I $FF_HOME hello_pipe.cpp -o hello_pipe -pthread`
  - On the Xeon PHI:
    - `g++ -std=c++11 -Wall -DNO_DEFAULT_MAPPING -O3 -I $FF_HOME hello_pipe.cpp -o hello_pipe -pthread`

# ClassWork1

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



# Class Work 2: using ff\_Pipe

- Simple file compressor using miniz.c:
  - The sequential implementation of the compressor is given (*simplecomp.cpp*) together with an utility program for decompressing the files (*compdecomp.cpp*).
  - The task is to modify the sequential code and implement a 3-stage pipeline version in which the first stage reads from the command line a list of files to compress, the second stage compresses each input file in memory and finally the third stage writes the compressed memory file into the disk (in a separate folder).
    - `g++ simplecomp.cpp -o simplecomp`
    - To decompress a file use the `compdecomp` program (first you have to compile the `compdecomp.cpp` file):
      - `./compdecomp d <compressed-file>`
  - All files needed are in the `~spm1501/public/ClassWork2` folder of the course machine