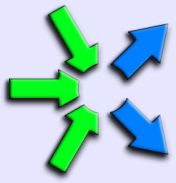
FastFlow introduction



Massimo Torquati

Computer Science Department, University of Pisa – Italy

torquati@di.unipi.it

SPM, October 2014

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 enough and flexible enough for programming multi/many-cores platforms
  - multi-core + GPGPUs + Xeon PHI + FPGA …..
- FastFlow has 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 will 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 >= 4.7)

The FastFlow tutorial

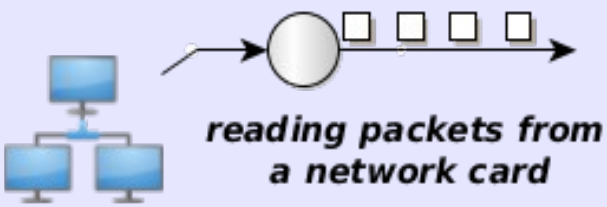
- The FastFlow tutorial is available as pdf file inside the FastFlow source code
  - It is also available on-line at the FastFlow web page
- All tests and examples described in the tutorial are available as a separate tarball file: **fftutorial_source_code.tgz**
- In the tutorial source code there are a number of very simple examples covering almost all aspects of using pipeline, farm, ParallelFor, map, mdf, etc..
  - Many (advanced) 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...
- Please start reading the simple tests, modifying and running them
- Then move to applications

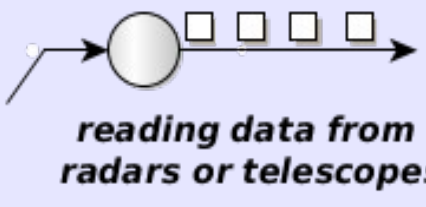
Stream concept (recap)

- 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.

Stream examples

- "*real streams*"

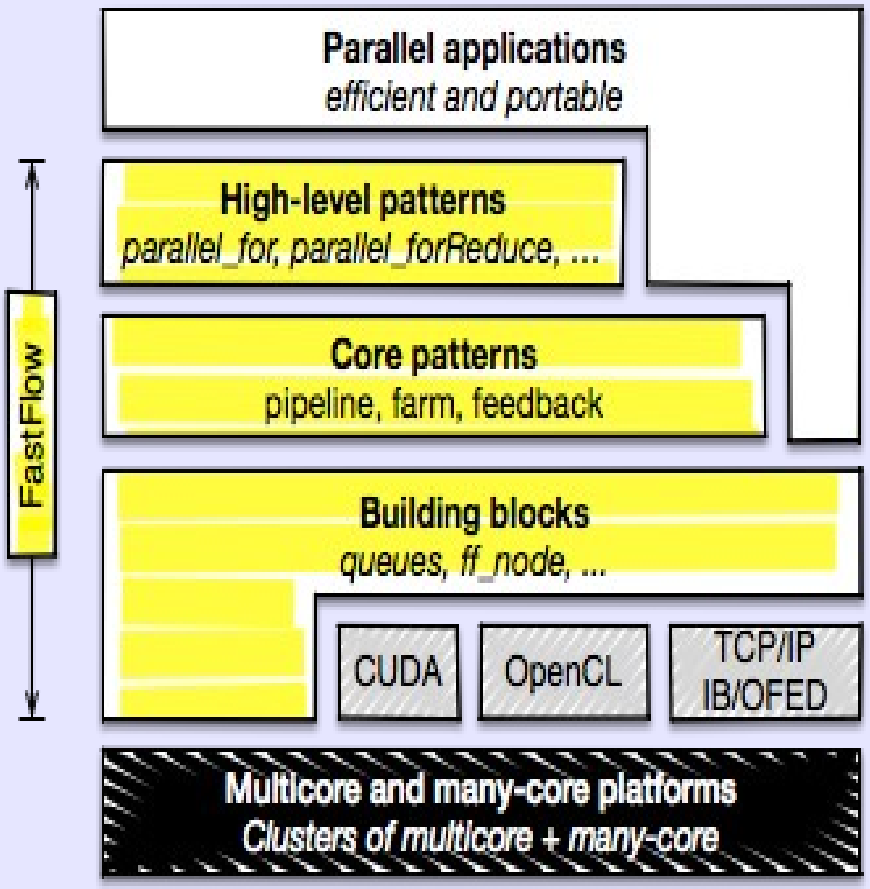




  - In these cases it is really important to satisfy minimum processing requirements (bandwidth, latency, etc...) in order to not lose data coming from the source
- "*fake streams*": streams produced by unrolling loops
  - The stream source is a software module

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

The FastFlow layers



- C++ class library
- **Promotes structured parallel programming**
- 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

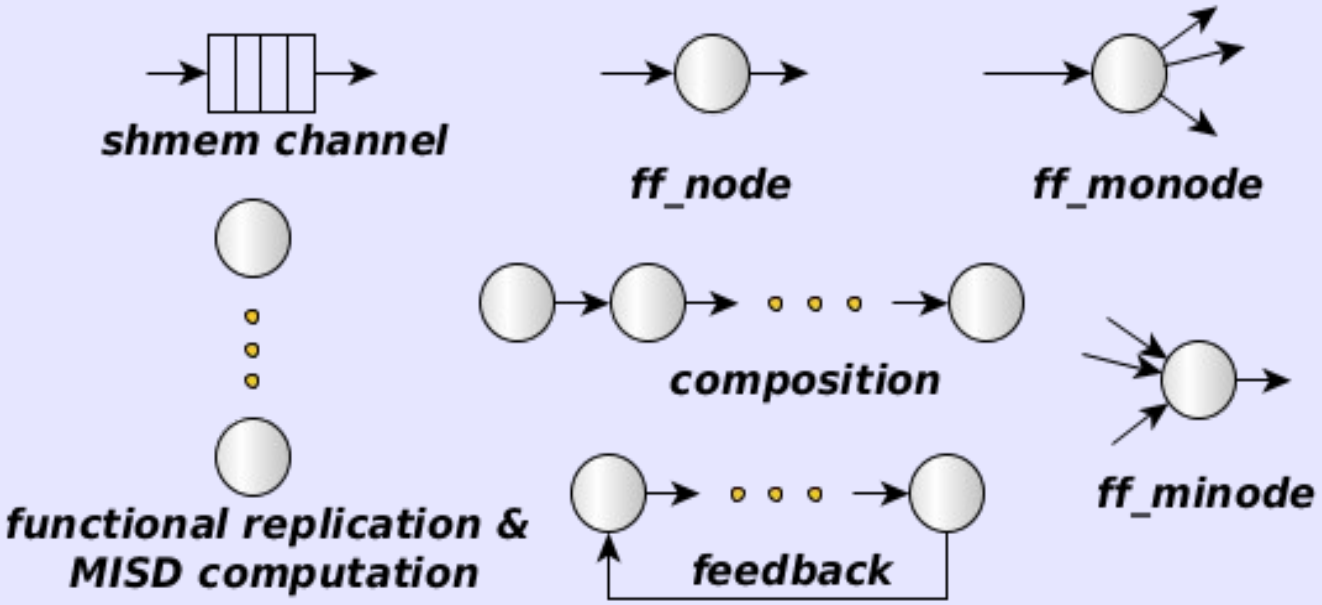
http://mc-fastflow.sourceforge.net
http://calvados.di.unipi.it/fastflow

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
- Task-parallel pattern

- 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

FastFlow lower level



- Minimal set of mechanisms and functionalities
- Nodes are concurrent entities (i.e. POSIX threads)
- Arrows are channels carrying pointers to data structures.
  - channels are SPSC lock-free queues
  - queues can be bounded or unbounded

Minimal definition of a node :

```
struct myNode: ff_node_t<myTask> {
  myTask *svc(myTask *task) { … return task; }
};
```

Stream parallelism in FastFlow

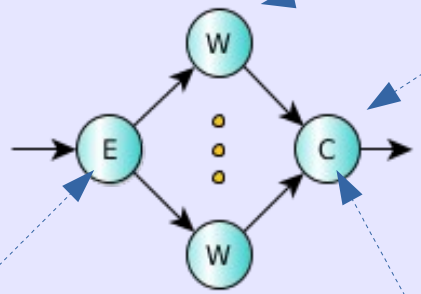
- FastFlow provides *pipeline* and *task-farm* patterns

pipeline


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

ff_node

task-farm



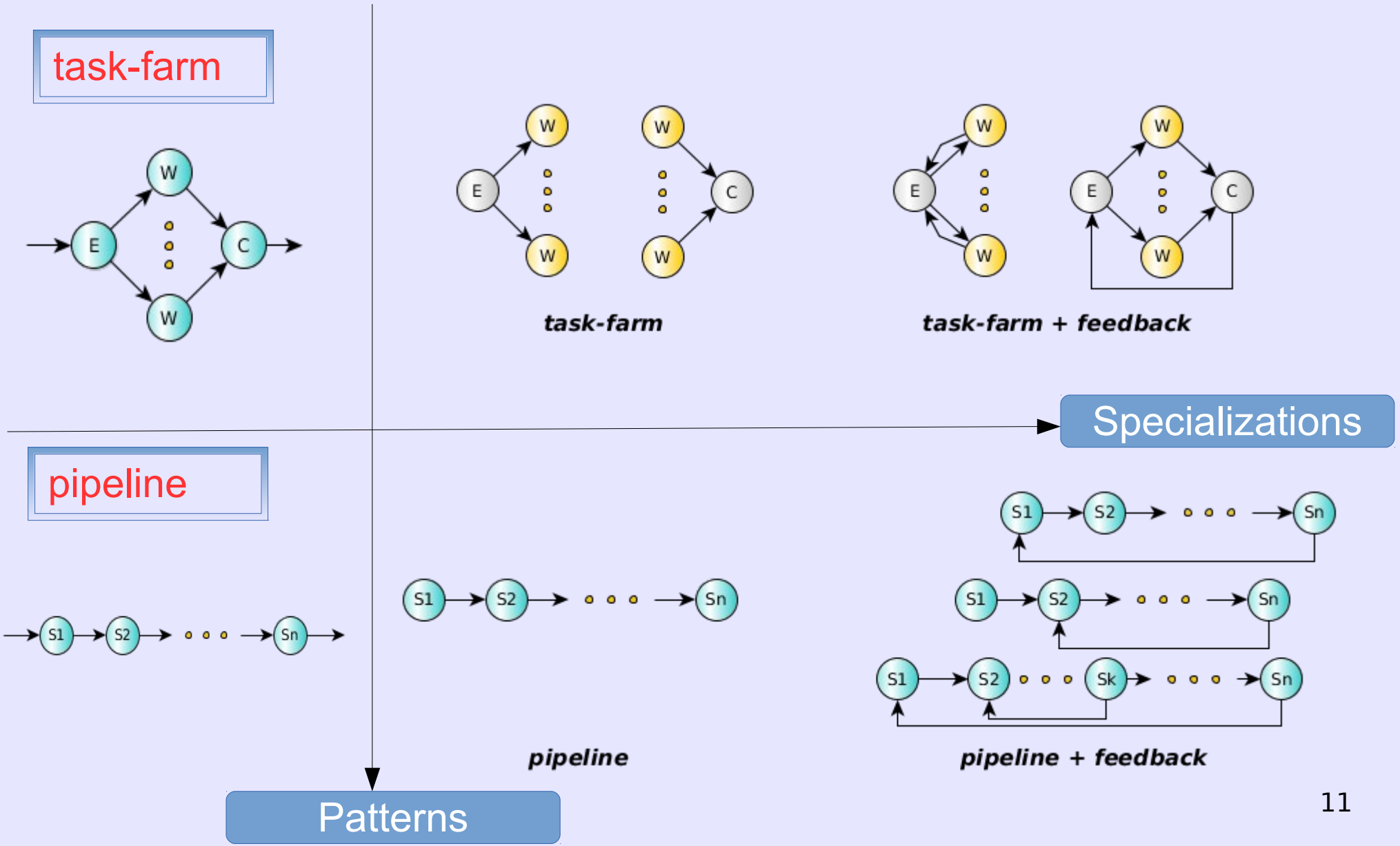

```
std::vector<ff_node*> WorkerArray;
prepareWorkerArray(WorkerArray);

ff_farm<> farm(WorkerArray, &E, &C);
farm.run_and_wait_end();
```

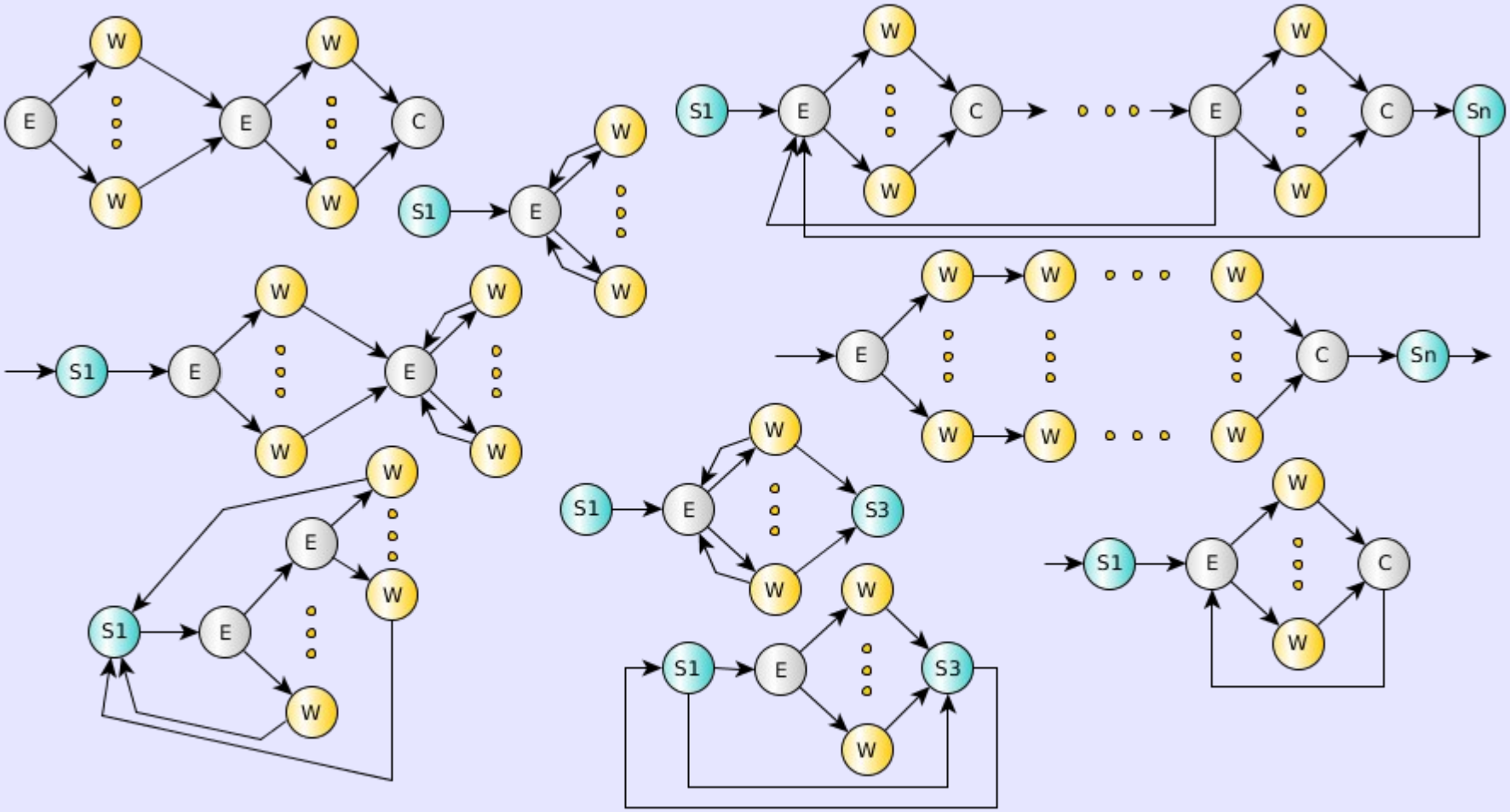
Emitter:
schedules input data items

Collector:
gathers results

Stream parallelism in FastFlow



Core Patterns Composition



pipeline + task-farm + feedback

Sequential node: ff_node_t

```
struct myNode: ff_node_t<myType> {

   int svc_init() {  // initialization management
    // called once when the thread starts (or restarts)
    return 0;  // <0 means error
   }

   myType* svc(myType *input) {
    // called each time an input task is available
    … // do something usefull here
    return output; // can also be EOS, GO_ON, ….
   }

   void svc_end() {   // termination management
    // called once when EOS is received in input or
    // if EOS is generated by the node itself
   }

};
```

- **A sequential *ff_node* is a 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 (**markers**):
  - ***EOS*** means End-Of-Stream
  - ***GO_ON*** means "I have no more tasks to send out, give me another input task (if any)"

Sequential node: generating and absorbing the stream of tasks

```
struct stageA: ff_node_t<myTask> {

   myType* svc(myType *input) {
      // generates N tasks and then EOS
      for(size_t i=0; i<N; ++i)
         ff_send_out(new myTask(....));
      return EOS;
   }
};
```

```
struct stageB: ff_node_t<myTask> {

   myType* svc(myType *input) {
      // do something with the input
      do_Work(input);
      return GO_ON; // no task is produced
   }
};
```

- Typically stageA is the first stage of a pipeline, it produces tasks by using the ff_send_out method or simply returning the task from the svc method
  - remember that if a stage does not receive inputs from other stages, the run-time keeps calling the svc (with nullptr as input) until EOS is produced
- Typically stageB is the last stage of a pipeline computation, it gets in input tasks without producing any outputs

FastFlow pipeline: ff_pipe

```
struct stageA: ff_node_t<myTask> {
    myType* svc(myType *) {
      for(size_t i=0; i<10; ++i)
        ff_send_out(new myTask(i));
      return EOS;
   }
};
struct stageB: ff_node_t<myTask> {
    myType* svc(myType *task) {
      return task;
   }
};
struct stageC: ff_node_t<myTask> {
    myType* svc(myType *task) {
      dowork(task);
      return GO_ON;
   }
};
stageA A; stageB B; stageC C;
ff_pipe<myTask> pipe(&A,&B,&C);
pipe.run_and_wait_end();
```

- Pipeline stages are ff_node(s)
- A pipeline itself is an ff_node
  - You may 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 (filter)
- Finally, the third stage applies the function dowork to each stream element and does not return any tasks

Simple examples

- Let's take a look at some very simple examples
- Tutorial simple tests (the ones inside the 'fftutorial_source_code/tests' dir):
  - hello_pipe.cpp
  - hello_pipe2.cpp
  - hello_node.cpp (if enough time)

FastFlow farm: ff_farm

```
struct Worker: ff_node_t<myTask> {
    myType* svc(myType *in) {
      F(in);
      return GO_ON
  }
};

std::vector<ff_node*> Workers;
Workers.push_back(new Worker);
Workers.push_back(new Worker);
ff_farm<> myFarm(Workers);
// myFarm.remove_collector();

ff_pipe<myTask>  pipe(&_1, &myFarm, ….);
pipe.run_and_wait_end();
```

- Farm's workers are ff_node(s) provided via an std::vector
- By providing different ff_node(s) it is easy to build a MISD farm
- By default the farm has an Emitter and a Collector, the Collector can be removed using:
  - myFarm.**remove_collector();**
- Emitter and Collector may be redefined by providing suitable ff_node objects
- Default task scheduling is **pseudo** round-robin
- Auto-scheduling:
  - myFarm.**set_scheduling_ondemand()**
- Possible to implement user's specific scheduling strategies (**ff_send_out_to**)
- Farms and pipeline can be nested and composed in any way

Simple ff_farm examples

- Let's comment on the code of the 2 simple tests presented in the FastFlow tutorial:
  - hello_farm.cpp
  - hello_farm2.cpp

- Then, let's take a look on how to define Emitter an Collector in a farm:
  - hello_farm3.cpp

- A farm in a pipeline without the Collector:
  - hello_farm4.cpp

Class Work1 in FastFlow

- http://didawiki.cli.di.unipi.it/doku.php/magistraleinformaticanetworking/spm/spm14exe1

- Simple pipeline (different versions):
  - pipe_square.cpp (pipeline implementation)
  - farm_square.cpp (mapreduce-like implemented as a farm)
  - parfor_square.cpp (mapreduce using the ParallelFor – not yet described)

- Sample map:
  - FastFlow implementation left as an exercise.