

Introduction to FastFlow programming

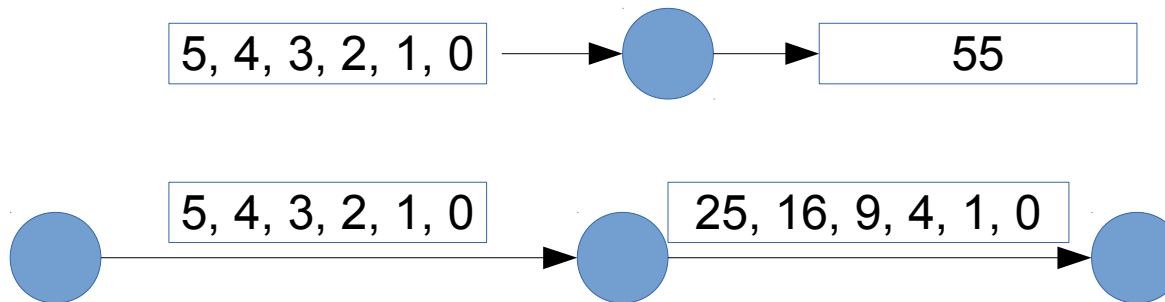
SPM lecture, November 2015

Massimo Torquati <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy

ClassWork1: comments

- Computing the sum of the square of the first N numbers 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;
};
```

```
// 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};
};
```

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

Core patterns: *ff_farm*

(1)

task-farm pattern

```
struct myNode: ff_node_t<myTask> {  
    myTask *svc(myTask * t) {  
        F(t);  
        return GO_ON;  
    }  
};  
  
std::vector<std::unique_ptr<ff_node>> W;  
W.push_back(make_unique<myNode>());  
W.push_back(make_unique<myNode>());  
  
ff_Farm<myTask>  
    myFarm(std::move(W));  
  
ff_Pipe<myTask>  
    pipe(_1, myFarm, <...other stages...>);  
  
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 (each worker computes a different function)
- 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();`
- Possibility to implement user's specific scheduling strategies (`ff_send_out_to`)
- Farms and pipelines can be nested and composed in any way

Core patterns: *ff_farm*

(2)

task-farm pattern

```
myTask *F(myTask * t,ff_node*const) {  
  .... <work on t> ....  
  return t;  
}
```

```
ff_Farm<myTask> myFarm(F, 5);
```

```
myTask *F(myTask * t,ff_node*const) {  
  .... <work on t> ....  
  return t;  
}
```

```
ff_OFarm<myTask> myFarm(F, 5);
```

- Simpler syntax
- By providing a function having a suitable signature together with the number of replicas
 - 5 replicas in the code aside
- Default scheduling or auto-scheduling

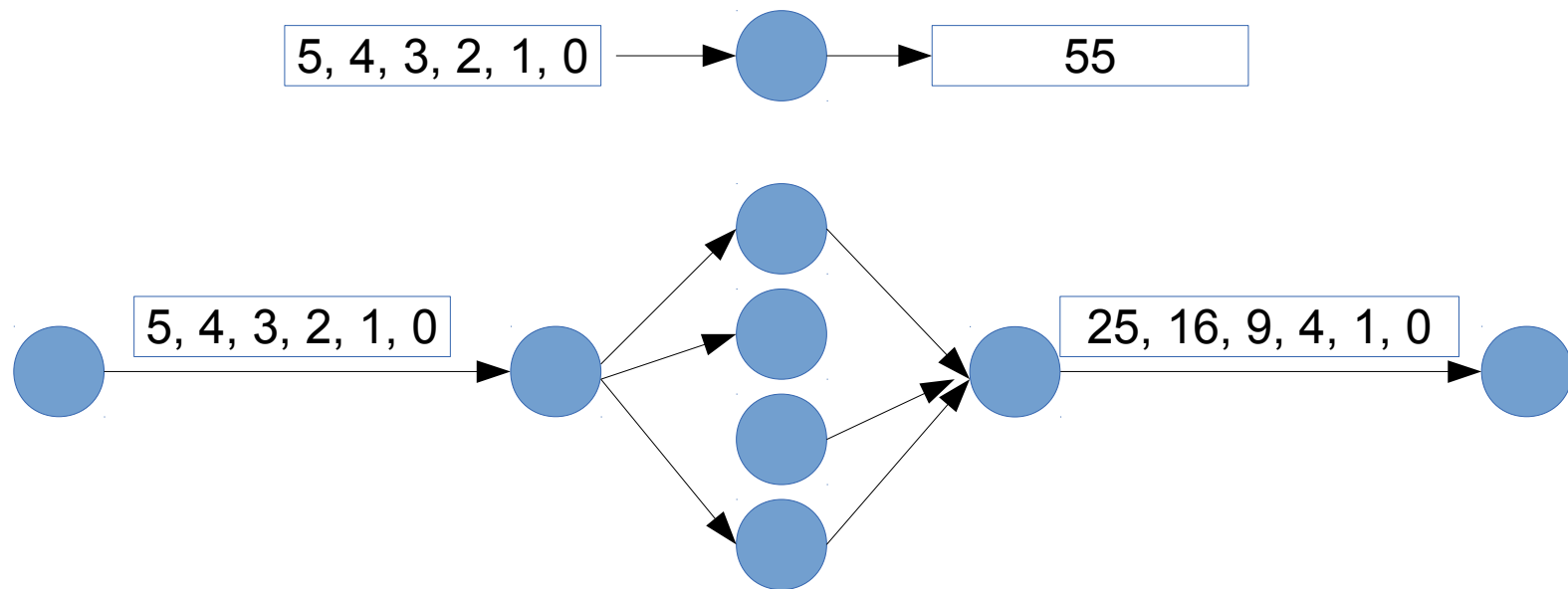
- Ordered task-farm pattern
- Tasks are produced in output in the same order as they arrive in input
- In this case it is not possible to redefine the scheduling policy

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 at how to define Emitter and Collector in a farm:
 - `hello_farm3.cpp`
- A farm in a pipeline without the Collector:
 - `hello_farm4.cpp`

ClassWork2

- Considering again the ClassWork1. Then, transform the middle stage of the pipeline in a task-farm.



- When it works, then try to remove the collector from the farm.

Class Work 3: using ff_Pipe and ff_Farm

- 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>`
 - Then implement the second stage by using an `ff_Farm`
 - All files needed are in the `~spm1501/public/ClassWork3` folder of the course machine