Introduction to FastFlow programming

SPM lecture, December 2017

Massimo Torquati <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy

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:

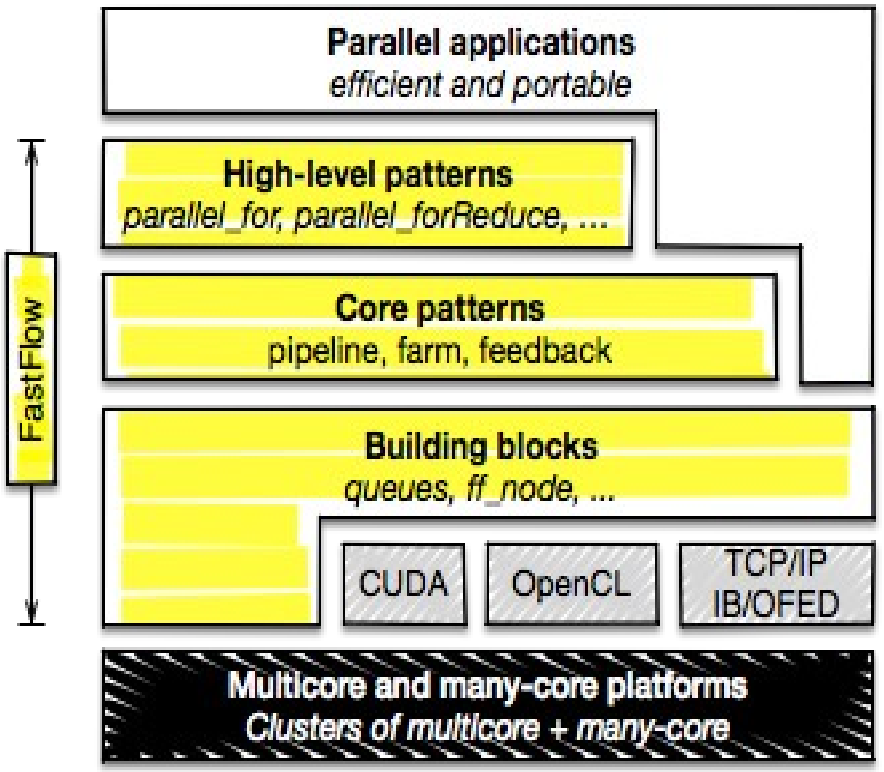
```
svn update
```

- The run-time (i.e. all you need for compiling your programs) is in the *ff* folder (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 on the FastFlow home page under “Tutorial”
  - http://mc-fastflow.sourceforge.net (aka 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

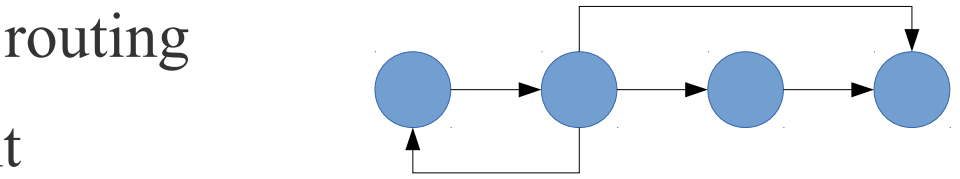


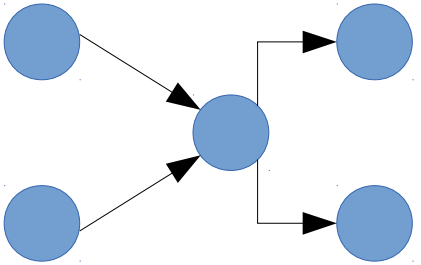
- C++ class library
- Promotes (high-level) structured parallel programming
- Streaming natively supported
- It aims to be flexible and efficient enough to target **multi-core**, **many-core** and **distributed heterogeneous systems**.
- Layered design:
  - **Building blocks** minimal set of mechanisms: 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

The FastFlow concurrency model

- Data-Flow programming model implemented via shared-memory
  - Nodes are parallel activities. Edges are true data dependencies
  - Producer-Consumer synchronizations
  - More complex synchronizations are embedded into the patter behaviour
  - Data is not moved/copied if not really needed
- Full user's control of message routing
- Non-determinism management





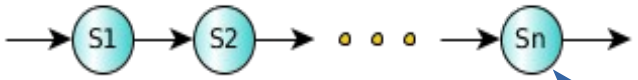
What FastFlow provides

- 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, D&C)

- 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

Stream Parallel Patterns in FastFlow ("core" patterns)

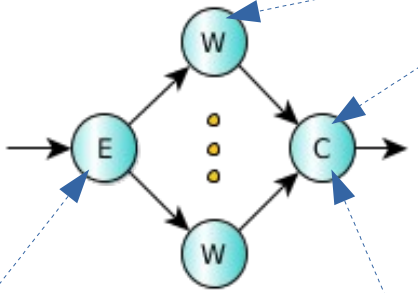
pipeline




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

task-farm

ff_node



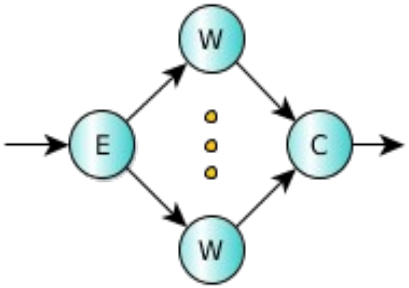

```
std::vector<std::unique_ptr<ff_node> >  Warray;

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

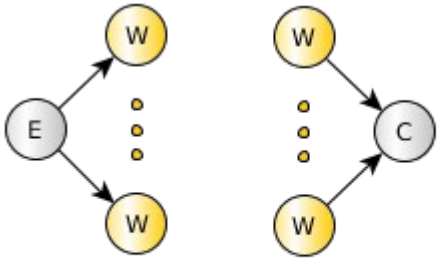
Emitter:
schedules input data items

Collector:
gathers results

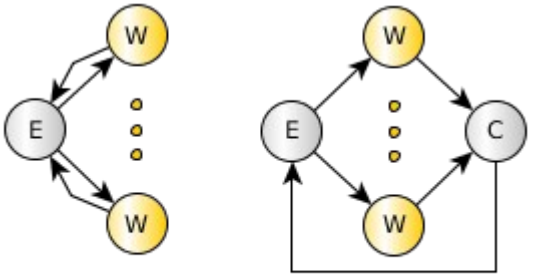
Stream Parallel Patterns (“core” patterns)

task-farm





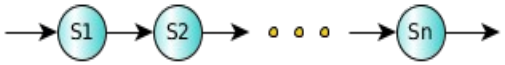
task-farm

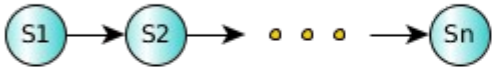


task-farm + feedback

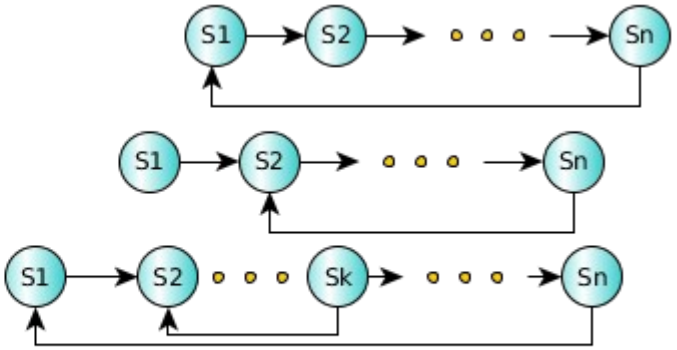
Specializations

pipeline





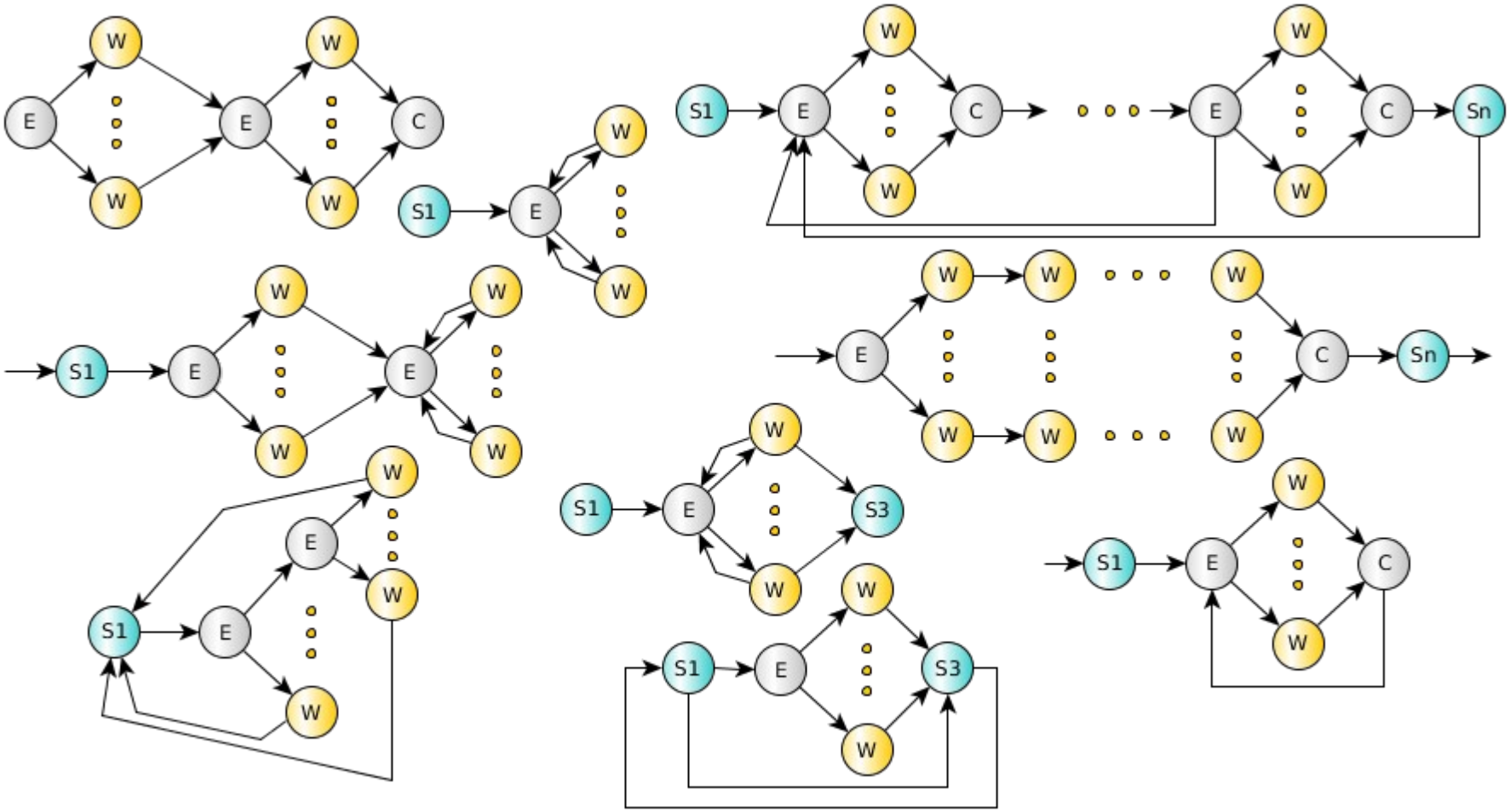
pipeline



pipeline + feedback

Patterns

Core patterns composition



pipeline + task-farm + feedback

High-Level Patterns

- Address application programmers' needs
- All of them are implemented on top of “core” patterns
  - Stream Parallelism: Pipe, Farm
  - Data Parallelism: Map, IterativeStencilReduce
  - Task Parallelism: PoolEvolution, MDF, TaskF, D&C
  - Loop Parallelism: ParallelFor, ParallelForReduce

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) { // mandatory
    // do something on the input task
    // called each time a task is available
    return task; // also EOS, GO_ON, ....
  };
  void svc_end() { // optional
    // 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 data items
  - 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
  };
};
```

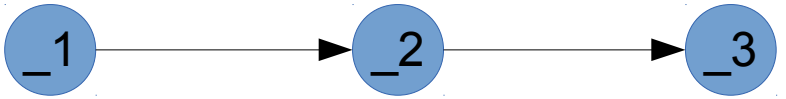
- Typically myNode1 is the first stage of a pipeline, it produces tasks by using the ***ff_send_out*** method or simply returning task from the svc method
- Typically myNode2 is the last stage of a pipeline computation, it gets in input tasks without producing any outputs

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* example

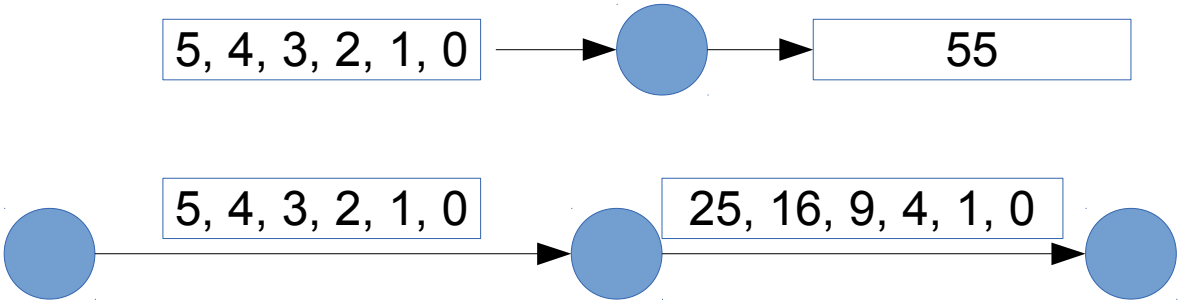
- Let's take a look at a simple test in the FastFlow tutorial:
  - hello_pipe.cpp

- 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 (before Knights Landing version):*
    - *g++ -std=c++11 -Wall **-DNO_DEFAULT_MAPPING -O3** -I $FF_HOME hello_pipe.cpp -o hello_pipe -pthread*

Simple pipeline example (*square*)

- 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);
```

- 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

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

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

- 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

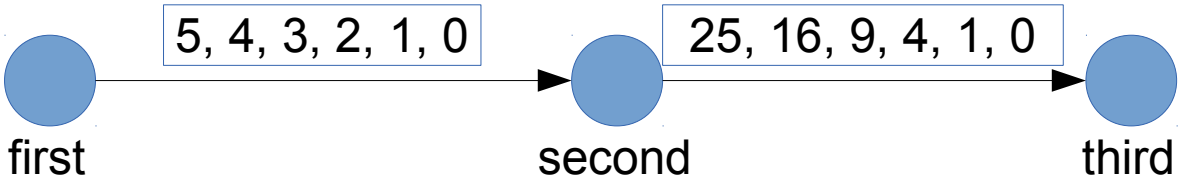
Simple *ff_farm* examples

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

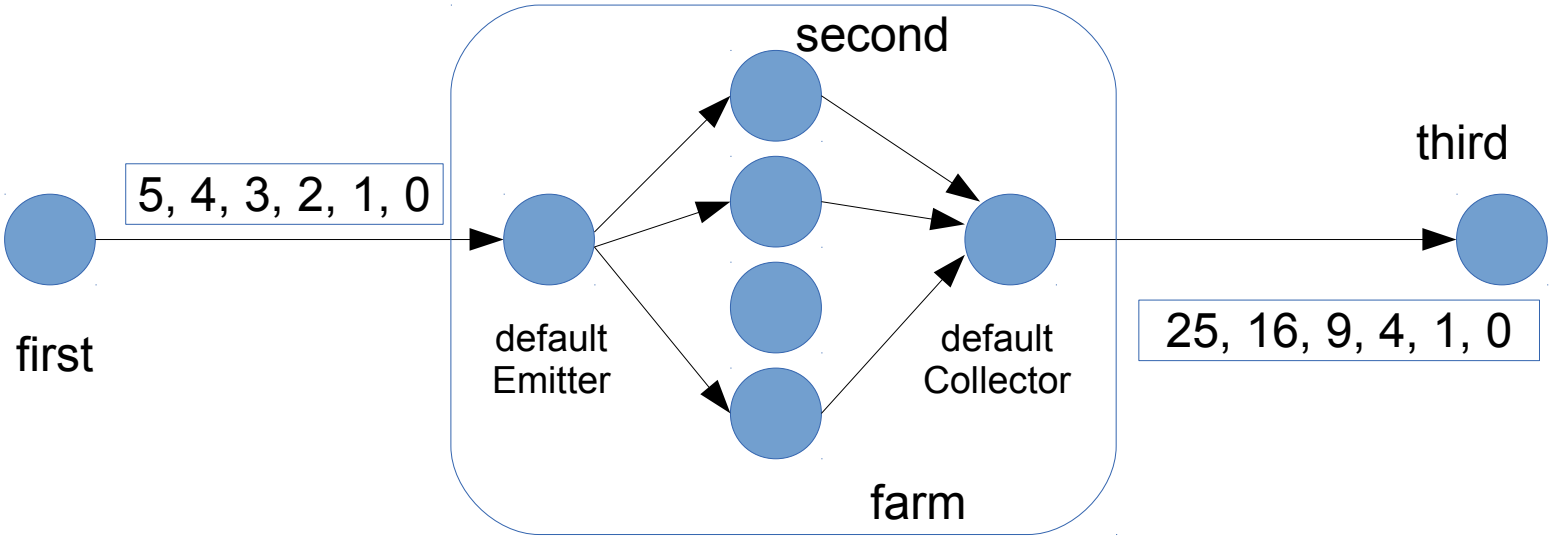
- Then, let's take a look at how to define Emitter an Collector in a farm:
  - hello_farm3.cpp
- A farm in a pipeline without the Collector:
  - hello_farm4.cpp

square example revisied (1)

- Let's consider again the *square* example: pipe(seq, seq, seq)



- 3-stage pipeline: pipe(seq, farm, seq)

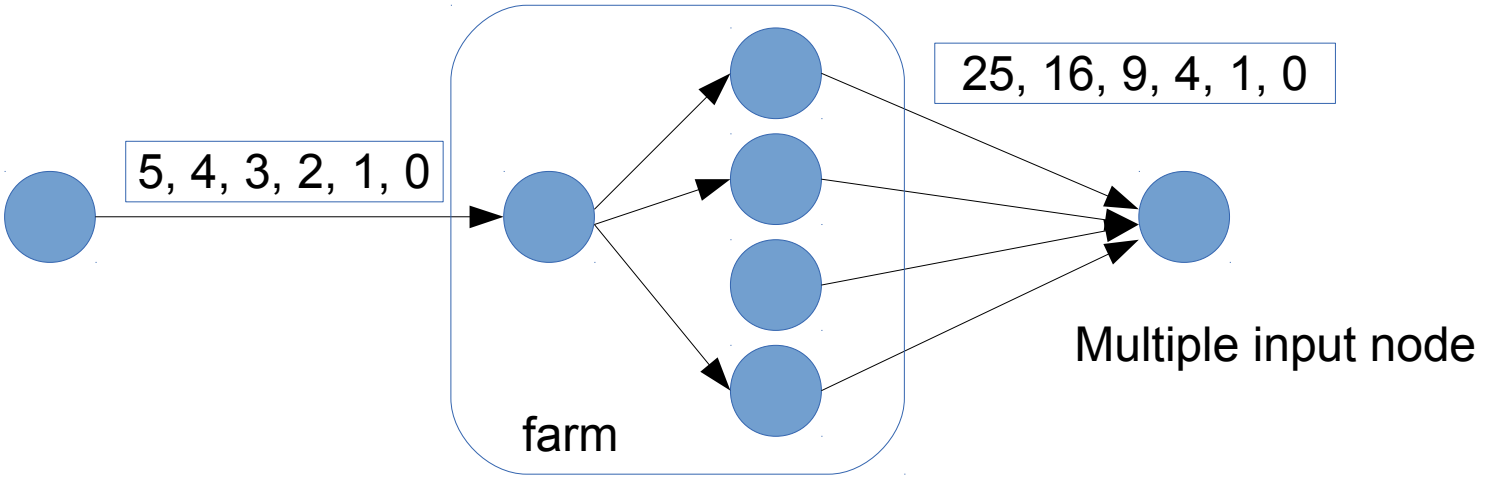


- Default task scheduling is (*pseudo) round-robin*
- The task collection in the Collector thread is "from any" (input non-determinism)

- Let's have a look at the code *farm_square1.cpp*

square example revisied (2)

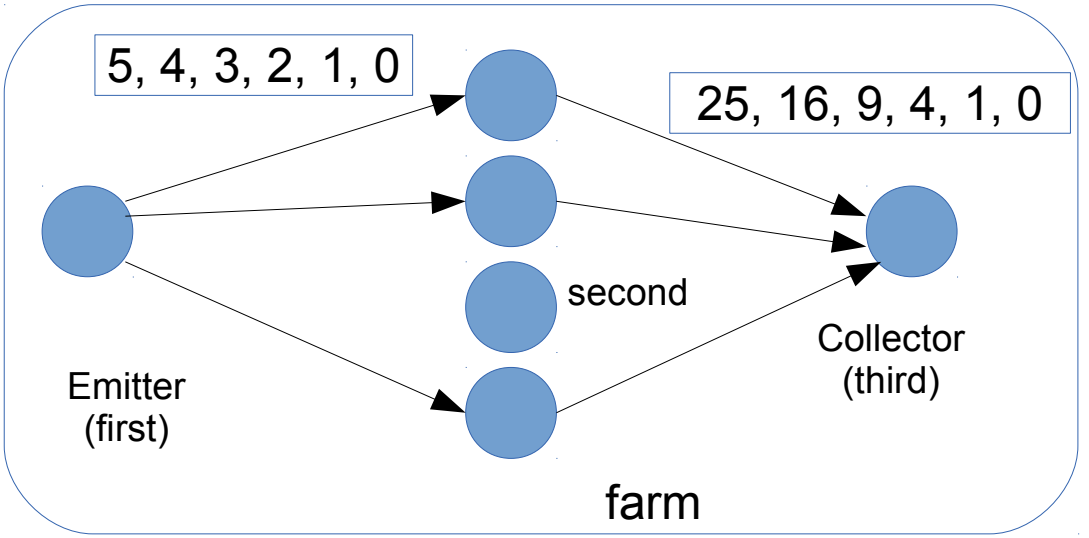
- 3-stage pipeline: pipe(seq, farm, seq)
- The farm does not have the collector node
- The third stage of the pipeline is a multi-input node (ff_minode_t)



- The Collector can be removed using:
  - myFarm.remove_collector();
  - If the next stage after the farm is a sequential node, it must be defined as
  - *ff_minode_t* (multi-input node)
- Let's see the *farm_square2.cpp* file

square example revisied (3)

- single farm with specialized Emitter and Collector: farm(seq, nw)



- The farm collector by default acts as a multi input node
- The farm emitter by default acts as a multi output node

- Let's see the farm_square3.cpp file

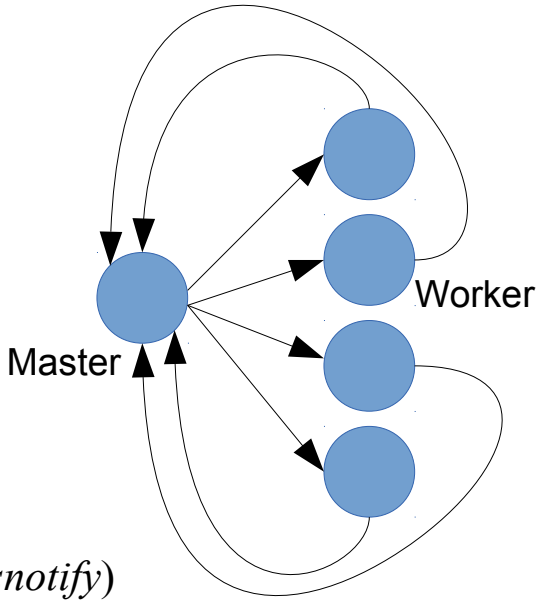
Ordered farm *ff_ofarm*

- Provides a total ordering between input and output
  - use case example: video streaming
- Limitations:
  - The number of tasks produced in output by the workers must be exactly the same of the number of tasks received in input
  - It is not possible to define your own scheduling and gathering policies
- If you don't need a strict input/output ordering then it is generally better to implement your own policy by re-defining the Emitter and the Collector

- Considering again the ClassWork2, try to replace the ff_Farm with the ff_OFarm in all examples (pay attention to the ff_OFarm class interface for the farm_square3.cpp version)

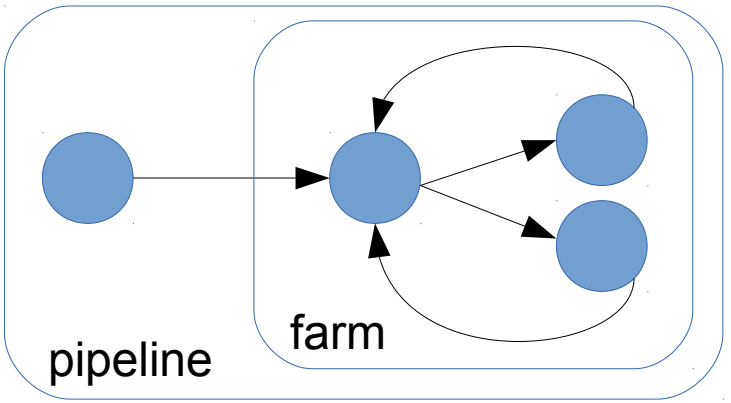
More on the *ff_farm*

- Auto-scheduling:
  - myFarm.set_scheduling_ondemand(<optional-value>)
- Possibility to implement user's specific scheduling strategies (ff_send_out_to)
  - *ff_send_out_to.cpp* example in the tutorial tests
- Master-Worker computation:
  - farm without the collector node
  - Workers send the results back to the Emitter



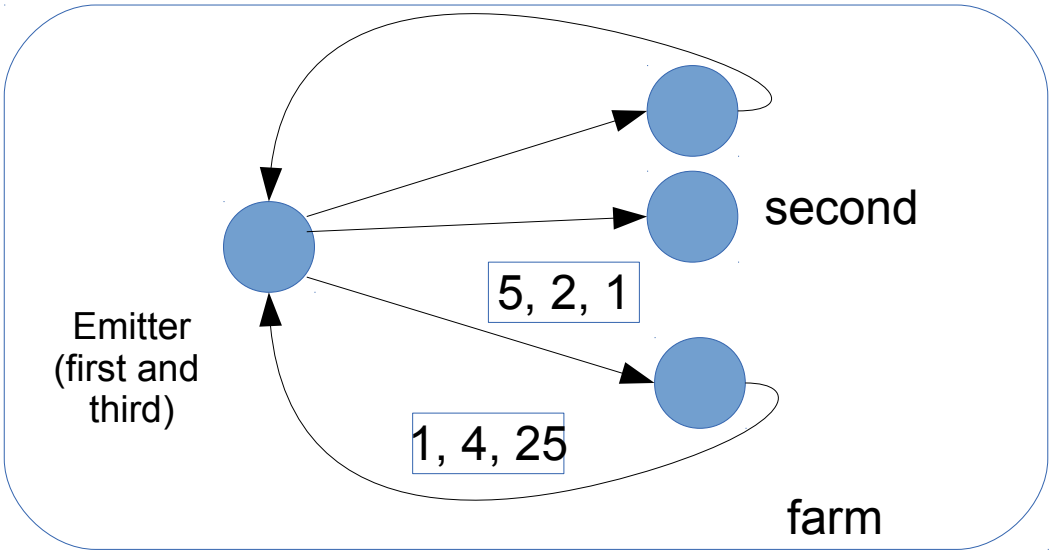
master-worker skeleton

- Let's see the *feedback.cpp* example in the tutorial tests
  - The termination protocol is a bit more complex (you need to use *eosnotify*)



FastFlow farm (again classWork2)

- Master-worker version:



- Let's see the farm_square4.cpp file

Data Parallel Computations

- In data parallel computations, large data structures are partitioned among the number of concurrent resources each one computing the same function (F) on the assigned partition
- Input data may come from an input stream
- Typically the function F may be computed independently on each partition
  - There can be dependencies as in stencil computations
- **Goal**: reduce the *completion time* for computing the input task
- Patterns:
  - map, reduce, stencil, scan,… typically they are encountered in sequential program as *loop-based computations*
- In FastFlow we have a high-level pattern for parallel-loop computations: **ParallelFor/ParallelForReduce**

FastFlow ParallelFor

- The ParallelFor patterns can be used to parallelize loops with independent iterations
- The class interface is defined in the file *parallel_for.hpp*
- Example:

```
// A and B are 2 arrays of size N

for(long i=0; i<N; ++i)
    A[i] = A[i] + B[i];
```

```
#include <ff/parallel_for.hpp>
using namespace ff;

ParallelFor pf(8);   // defining the object

pf.parallel_for(0, N, 1, 0, [&A,B](const long i) {
    A[i] = A[i] + B[i];
}, 4);
```

- Constructor interface (all parameters have a default value):
  - ***ParallelFor(maxnworkers, spinWait, spinBarrier)***
- parallel_for interface (on the base of the number and type of bodyF arguments you have different parallel_for methods):
  - ***parallel_for(first, last, step, chunk, bodyF, nworkers)***
  - ***bodyF is a C++ lambda-function***

FastFlow ParallelForReduce

- The ParallelForReduce patterns can be used to parallelize loops with independent iterations having reduction variables (map+reduce)
- Example:

```
// A is an array of long of size N
long sum = 0;
for(long i=0; i<N; ++i)
    sum += A[i];
```

```
#include <ff/parallel_for.hpp>
using namespace ff;

ParallelForReduce<long> pfr;
long sum=0;
pfr.parallel_reduce(sum, 0,
                    0,N,1,0, [](const long i, long &mysum) {
                mysum += A[i] + B[i];
    },
    [ ](long &s, const long e) { s += e;}
);
```

- The constructor interface is the same of the ParallelFor (but the template type)
- parallel_reduce method interface
  - ***parallel_reduce(var, identity-val, first, last, step, chunk, mapF, reduceF, nworkers)***
  - ***mapF and reduceF are C++ lambda-functions***

ParallelForReduce *example*

- Dot product (or scalar product or inner product), takes to vectors (A,B) of the same length, it produces in output a single element computed as the sum of the products of the corresponding elements of the two vectors. Example:

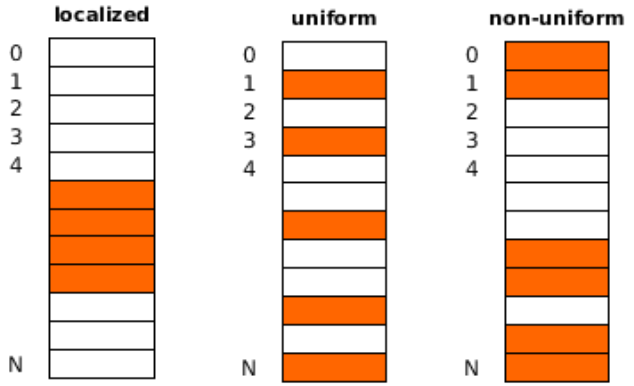
```
long s=0;
for(long i=0; i<N; ++i)  s += A[i] * B[i];
```

- Let's comment the FastFlow parallel implementation in the tutorial folder

  <fastflow-dir>/tutorial/fftutorial_source_code/examples/dotprod/dotprod.cpp

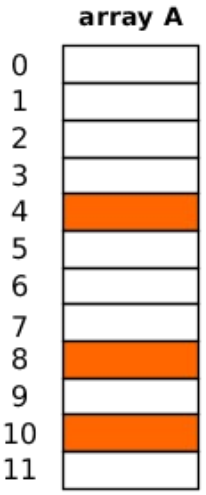
Iterations scheduling

- Suppose the following case:
- We have a computation on an array A of size N.
  - for(size_t i=0;i<N;++i) A[i]=F(A[i]); // map like computation
- You know that the time difference for computing different elements of the array A may be large.



- Problem: how to schedule loop iterations ?

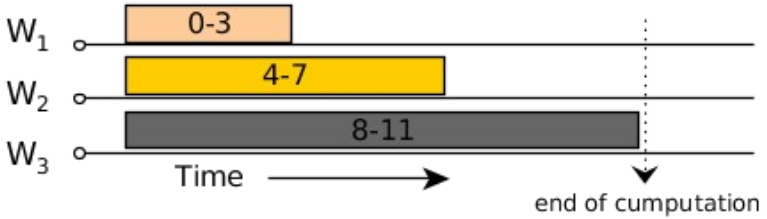
Iterations scheduling: example



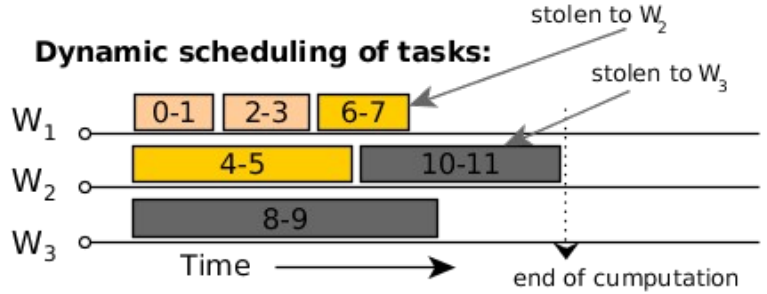
Suppose to have 3 workers and a chunksize=2, then the initial plan used for scheduling iterations is

wid	#tasks	min-max
0	2	0-3
1	2	4-7
2	2	8-11

Static assignment of tasks:



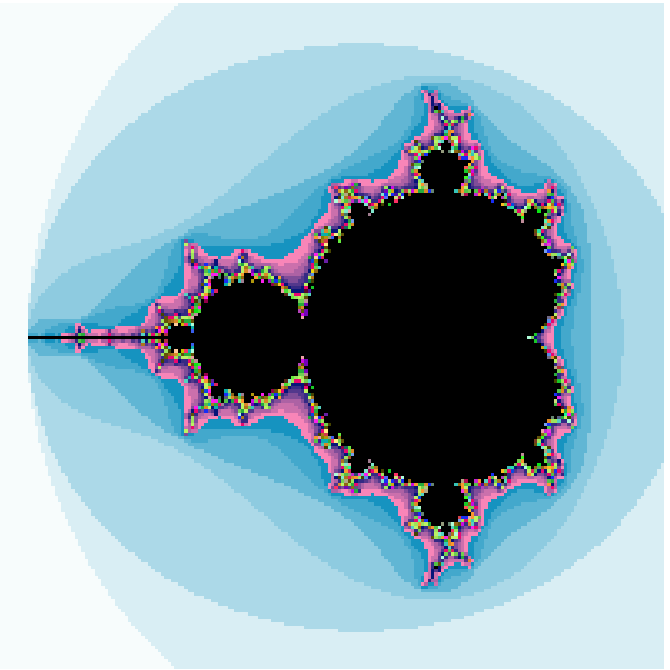
Dynamic scheduling of tasks:



Iterations scheduling in the ParallelFor* patterns

- Iterations are scheduled according to the value of the "*chunk*" parameter
  parallel_for(start, stop, step, ***chunk***, body-function);
- Three options:
  - chunk = 0 : static scheduling, at each worker thread is given a contiguous chunk of ~(#iteration-space/#workers) iterations
  - chunk > 0: dynamic scheduling with task granularity equal to the *chunk* value
  - chunk < 0: static scheduling with task granularity equal to the *chunk* value, chunks are assigned to workers in a round-robin fashion

Mandelbrot set example



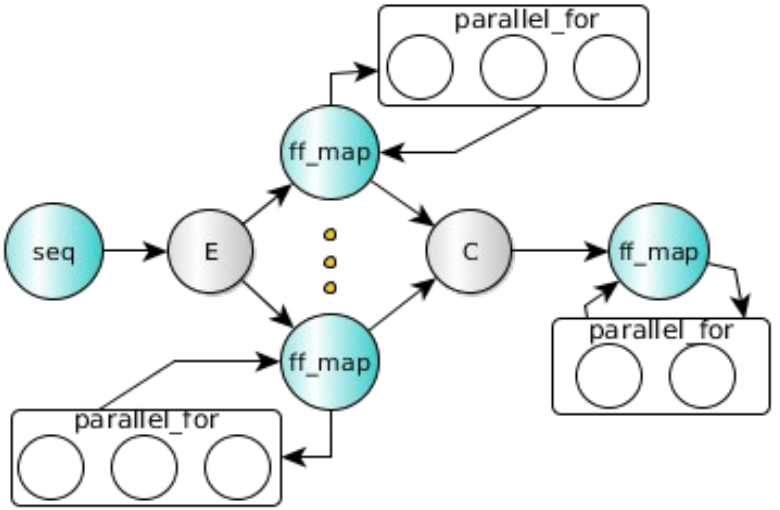
- Very simple data-parallel computation
  - Each pixel can be computed independently
  - Simple ParallelFor implementation
- Black-pixel requires much more computation
- A naïve partitioning of the images quickly
  leads to load unbalanced computation and poor performance
  - Let's consider as the minimum computation unit a single image row

    (image size 2048x2048, max 10^3 iterations per point)
    - ParallelFor Static partitioning of rows (48 threads) chunk=0 **MaxSpedup ~14**
    - ParallelFor Dynamic partitioning of rows (48 threads) chunk=1 **MaxSpeedup ~37**

  - <fastflow-dir>/tutorial/fftutorial_source_code/example/mandelbrot_dp/mandel.cpp

Combining Data Parallel and Stream Parallel Computations

- It is possible to nest data-parallel patterns inside a pipeline and/or a task-farm pattern



- We have mainly two options:
  - To use a ParallelFor* pattern inside the *svc* method of a FastFlow node
  - By defining a node as an ***ff_Map*** node

The ff_Map pattern

- The *ff_Map* pattern is just a ff_node_t that wraps a ParallelForReduce pattern

  ff_Map< Input_t, Output_t, reduce-var-type>

- Inside pipelines and farms, it is generally most efficient to use the ff_Map than a plain ParallelFor because more optimizations may be introduced by the run-time (mapping of threads, disabling/enabling scheduler thread, etc...)

- Usage example:

```
#include <ff/map.hpp>
using namespace ff;

struct myMap: ff_Map<Task,Task,float> {
    using map = ff_Map<Task,Task,float>;

    Task *svc(Task *input) {

        map::parallel_for(....);

        float sum = 0;
        map::parallel_reduce(sum, 0.0, ….);

        return out;
    }
};
```

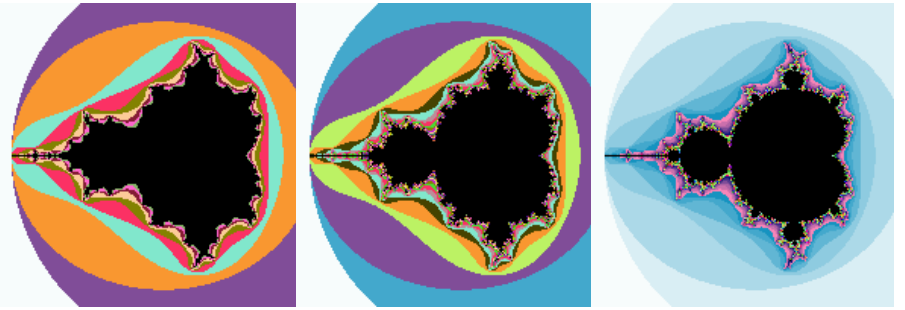
ff_Map *example*

- Let's have a look at the simple test case in the FastFlow tutorial

  <fastflow-dir>/tutorial/fftutorial_source_code/tests/hello_map.cpp

Mandelbrot set

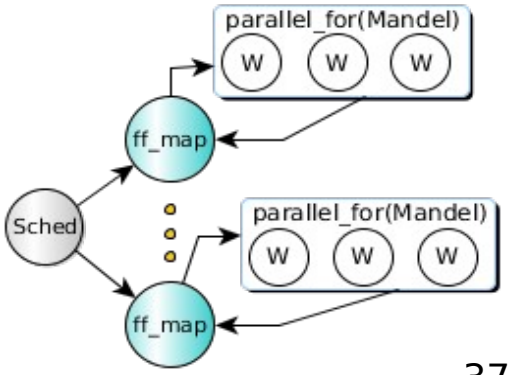
- Suppose we want to compute a number of Mandelbrot images (for example varying the computing threshold per point)

...  ...

- We have basically two options:
  1. One single parallel-for inside a sequential for iterating over all different threshold points
  2. A task-farm with map workers implementing two different scheduling strategies

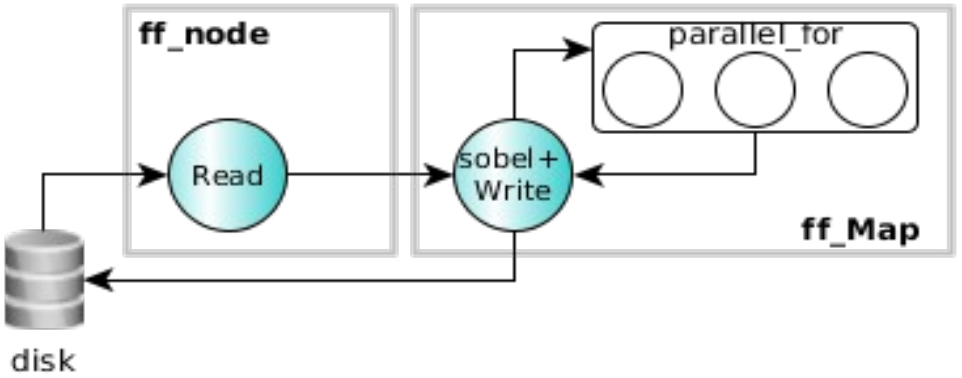
```
for_each threshold values
    parallel_for ( Mandel(threshold));
```

- Which one is better having limited resources ?
  - Depends on many factors, *too difficult to say in advance*

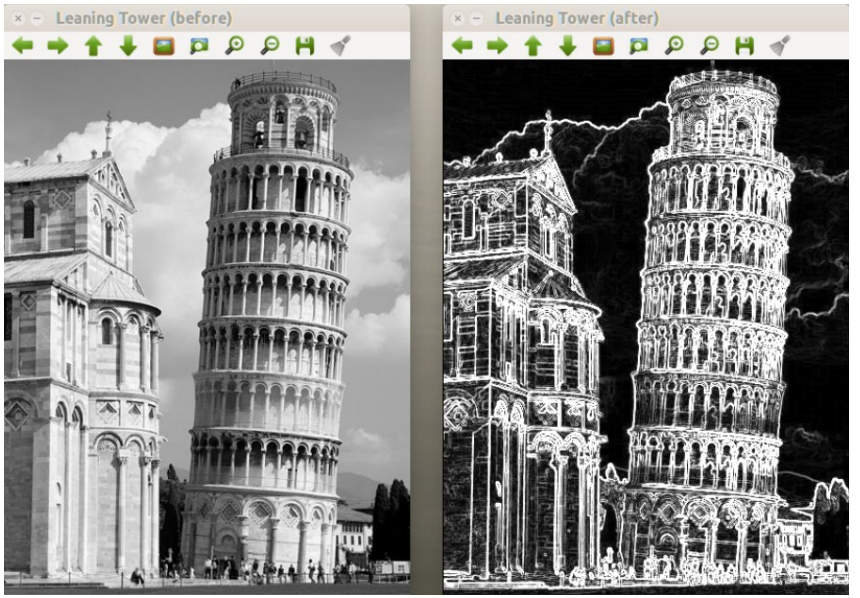


Moving quickly between the two solutions is the key point

Parallel Pipeline + Data Parallel : Sobel filter



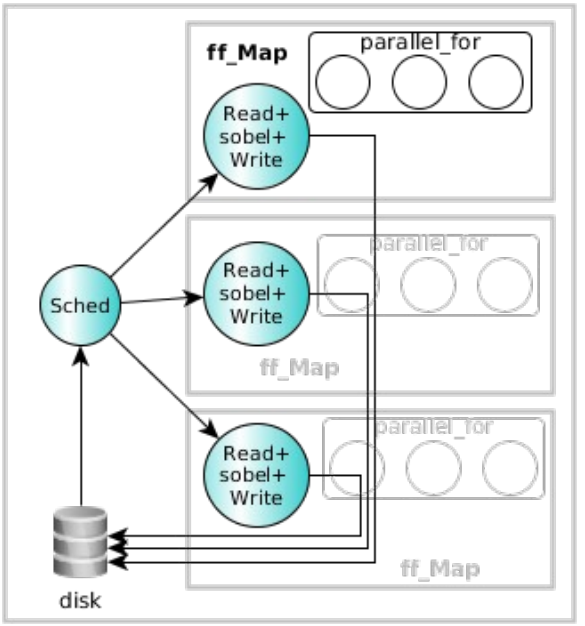

```
struct sobelStage: ff_Map<Task> {
   sobelStage(int mapwks):
      ff_Map<Task>(mapwrks, true) {};

   Task *svc(Task*task) {
     Mat src = *task->src, dst= *task->dst;
     ff_Map<>::parallel_for(1,src,src.row-1,
         [src,&dst](const long y) {
           for(long x=1;x<src.cols-1;++x) {
             ......
             dst.at<x,y> = sum;
           }
         });
     const std::string outfile="./out"+task->name;
     imwrite(outfile, dst);
   }
```



- The first stage reads a number of images from disk one by one, converts the images in B&W and produces a stream of images for the second stage
- The second stage applies the Sobel filter to each input image and then writes the output image into a separate disk directory

Parallel Pipeline + Data Parallel : Sobel filter

- We can use a task-farm of ff_Map workers
- The scheduler (Sched) schedules just file names to workers using an on-demand policy
- We have two level of parallelism: the number of farm workers and the number of map workers



- 2 Intel Xeon CPUs E5-2695 v2 @ 2.40GHz (12x2 cores)

- 320 images of different size (from few kilos to some MB)

- sobel (seq): ~ 1m

- pipe+map (4): ~15s

- farm+map (8,4): ~5s
- farm+map (32,1): ~3s