GridKa School 2014

Big Data, Cloud Computing and Modern Programming

Introduction to FastFlow programming

Hands-on session

Massimo Torquati <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy

Outline

- The FastFlow tutorial
- FastFlow basic concepts
  - stream concept
  - FF building blocks
- Core patterns:
  - *pipeline* & *task-farm*
- High-level patterns:
  - ParallelFor/ParallelForReduce/Map
  - Macro-DataFlow (mdf)

The FastFlow tutorial

- The FastFlow tutorial is available as pdf file on the GridKa wiki page in the "Programming Multi-Core using FastFlow" session
- All tests and examples described in the tutorial are available as a separate tarball file: **fftutorial_source_code.tgz**
  - can be downloaded from the wiki page)
- 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 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

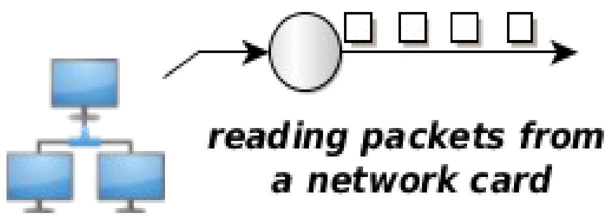
Let's start working!

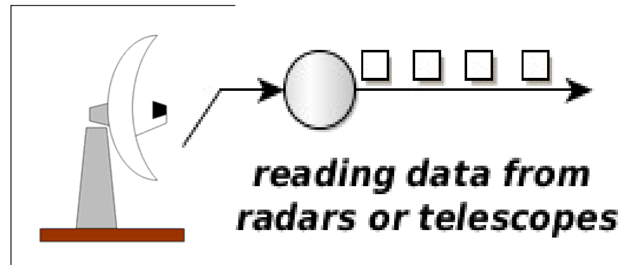
Stream concept

- 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“*



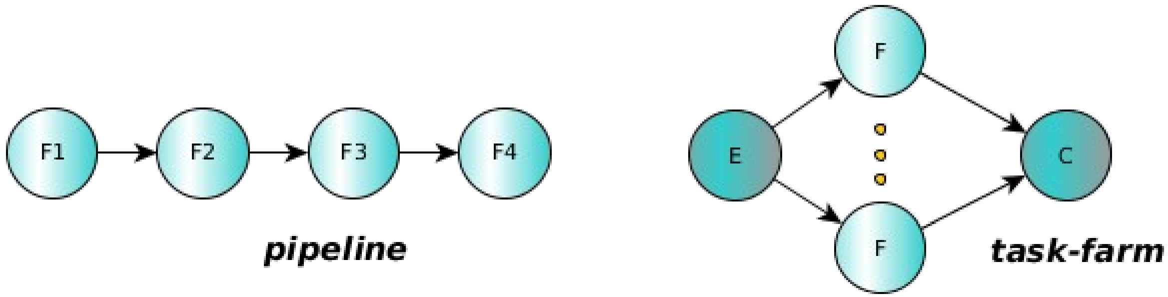


  - In these cases it is really important to satisfy minimum processing requirements (bandwidth, latency, etc...) in order to not lose data coming from the source

- “*fake streams*”: streams produced by unrolling loops
  - You don't have an “infinite“ source of data
  - The source is a software module

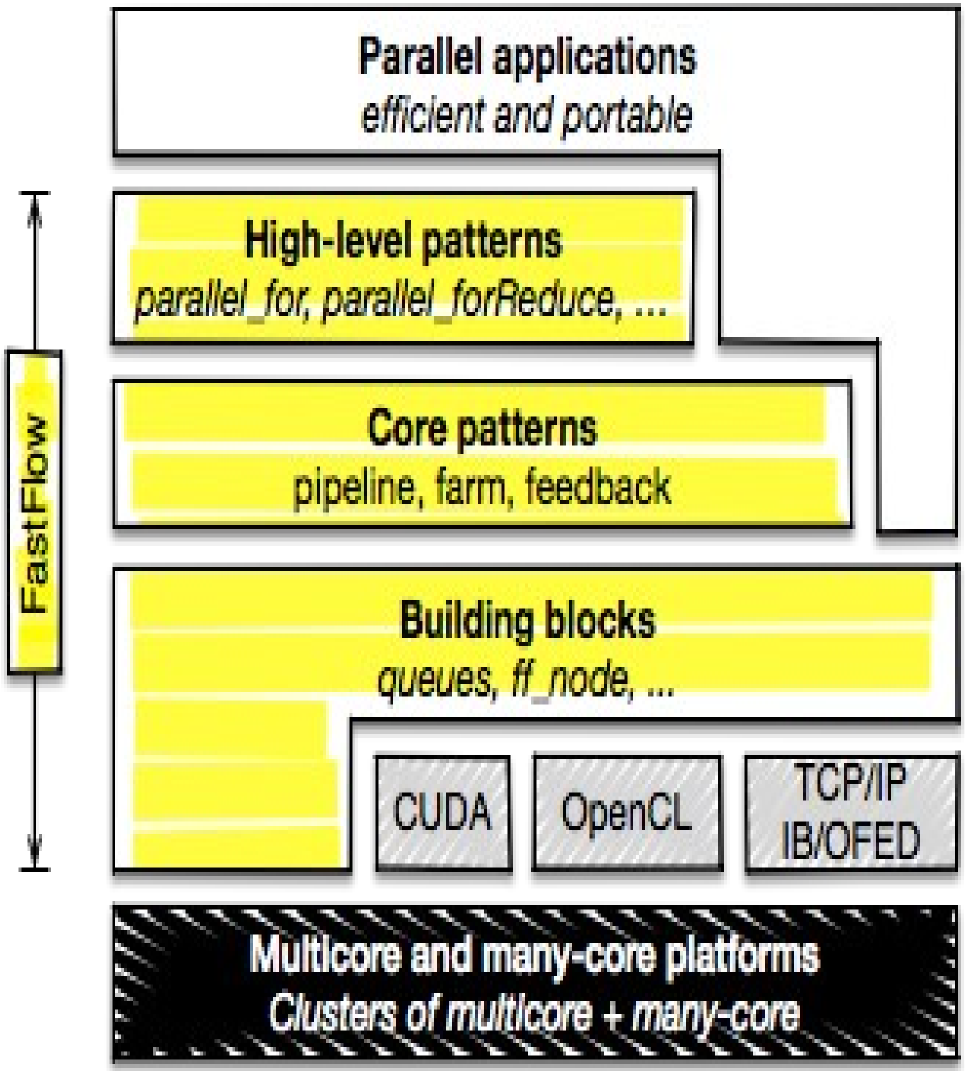
```
for(i=start; i<stop; i+=step)
   allocate data for a task
   create a task
   send out the task
```

Patterns operating on stream



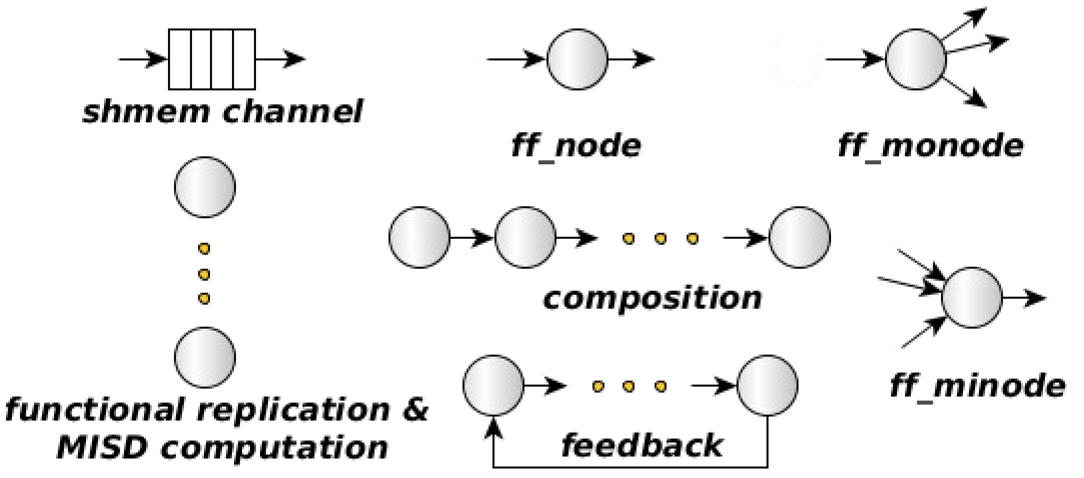
- ***pipeline**:* computes F4(F3(F2(F1(x)))) for each x
  - Pipeline computing elements are called *stages*
- ***task-farm*** (or *farm*), models *functional replication*
  - Sometimes called also “master-worker”
  - Computing elements called: Emitter (E), Worker (computing F) and Collector (C)
  - The Emitter, schedules tasks towards the Workers
  - The Collector, gathers tasks from Workers

The FastFlow layers



- C++ class library
- It 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

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

Core patterns: sequential *ff_node*

code wrapper pattern

```
struct myNode: ff_node {
  int svc_init() { // optional
    // called once for initialization purposes
    return 0;  // <0 means error
  }
  void *svc(void * 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 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:
  - ***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 {
  void *svc(void * 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 {
  void *svc(void * t) {
     // do something with the task
     Task *task=reinterpret_cast<Task*>(t);
     do_Work(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 {
  void *svc(void *) {
     for(long i=0;i<10;++i)
        ff_send_out(new myTask(i));
     return EOS;
}};
struct myNode2: ff_node {
  void *svc(void *task) {
     return task;
}};
struct myNode3: ff_node {
  void *svc(void * task) {
      f3((myTask*)task);
    return GO_ON;
}};
myNode1 _1;
myNode2 _2;
myNode3 _3;
ff_pipe<myTask> 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 comment on the code of the 2 simple tests presented in the FastFlow tutorial:
  - hello_pipe.cpp
  - hello_pipe2.cpp

Core patterns: *ff_farm*

task-farm pattern

```
struct myNode: ff_node {
    void *svc(void * t) {
      F(reinterpret_cast<Task*>(t));
    return GO_ON;
}};

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

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

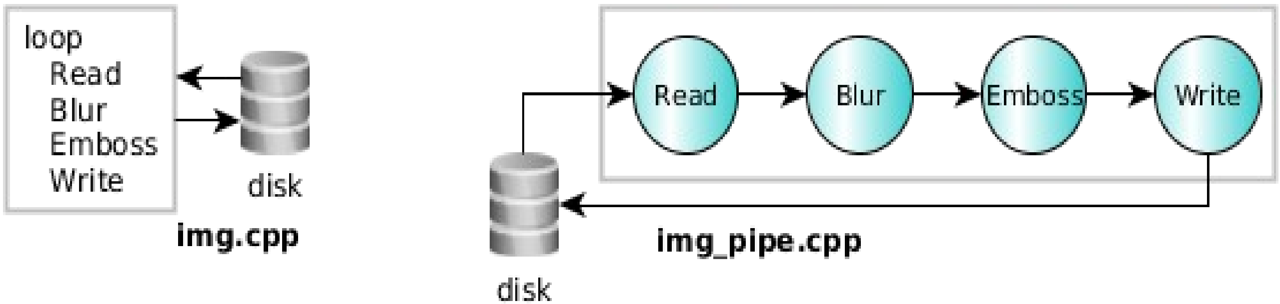
- Farm's workers are ff_node(s) provided via an std::vector
- By providing different ff_node(s) it is easy to build a MISD farm
- By default the farm has an Emitter and a Collector, the Collector can be removed using:
  - myFarm.**remove_collector();**
- Emitter and Collector may be redefined by providing 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

Examples: image filtering (img.cpp & img_pipe.cpp)




```
// 4-stage pipeline
ff_pipe<Task> pipe(new Read(filenames), BlurFilter, EmbossFilter, Write);
pipe.run_and_wait_end();
```

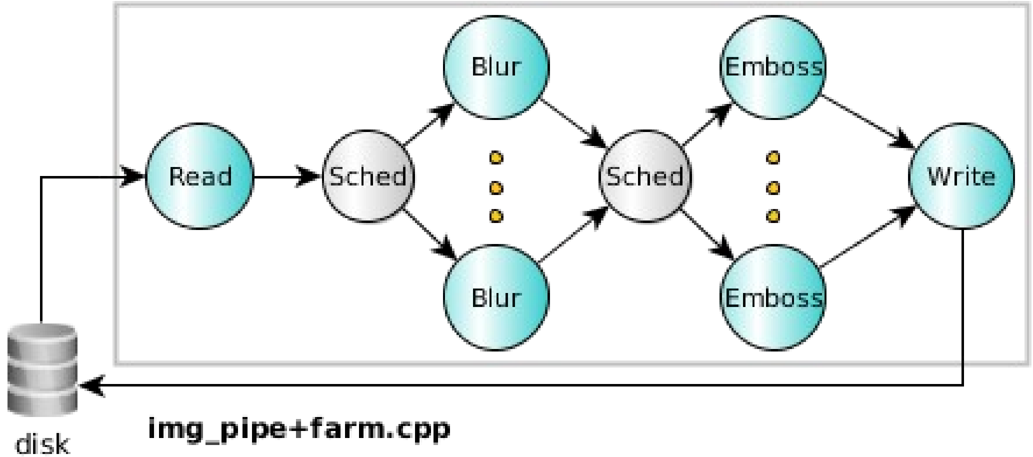
```
// 1st stage
struct Read: ff_node {
  void *svc(void *) {
    for(long i=0;i<num_images;++)
       Image *img = new Image;
       Img->read(filename);
       Task *task = new Task(img,filename);
       ff_send_out(task);
  }
  return EOS; // End-Of-Stream
};
```

```
// 2nd stage
Task *BlurFilter(Task *in, ff_node*const) {
    in->image->blur();   return in;
}
```

```
// 3rd stage
Task *EmbossFilter(Task *in, ff_node*const) {
    in->image->blur();     return in;
}
```

```
// 4th stage
Task *Write(Task *in, ff_node*const) {
    in->image->write(in->name);
    delete in->image;
    delete in;
    return (Task*)GO_ON;
}
```

Examples: image filtering (img_pipe+farm.cpp)



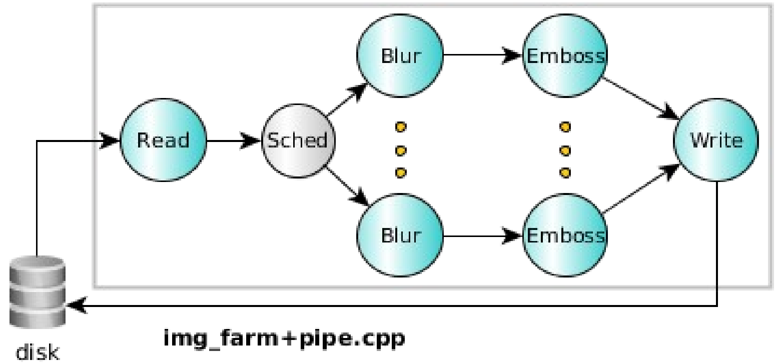

```
// 4-stage pipeline
ff_farm<> farmBlur(BlurFilter);
farmBlur.remove_collector();
ff_farm<> farmEmboss(EmbossFilter);
ff_pipe<Task> pipe(new Read(filenames), &farmBlur, &farmEmboss, Writer);
pipe.run_and_wait_end();

// ff_node wrapper to the Write function
struct Writer: ff_minode {
  void *svc(void *task) {
     return  Write(reinterpret_cast<Task*>(task), this);
};
```

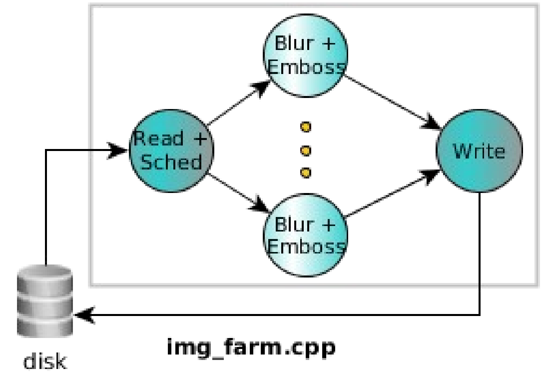
Other nodes are the same as before

Examples: image filtering

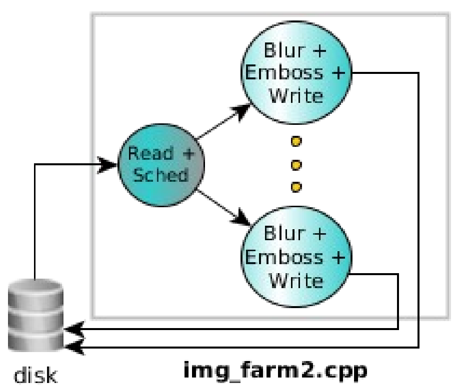
Other simple transformations



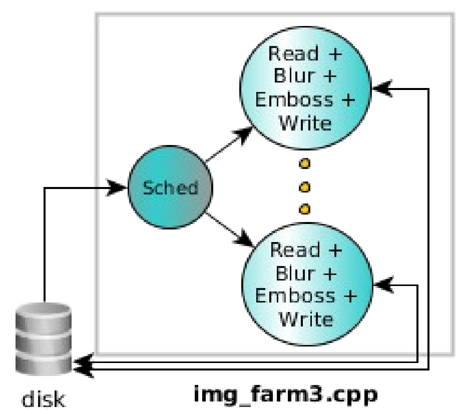
img_farm+pipe.cpp



img_farm.cpp



img_farm2.cpp



img_farm3.cpp

Let's see the code and how It works !

Proposed exercises using ff_pipe & ff_farm

- Simple file compressor using miniz.c:
  - The sequential implementation is given ***simplecomp.cpp***
  - The task is to implement both a pipeline implementation and a task-farm implementation of the same code.
    - simplecomp_pipe.cpp
    - simplecomp_farm.cpp
  - **HINT**: the structure is quite similar to img_pipe.cpp and img_farm.cpp, respectively.

- A more complex and efficient implementation is left as homework

- One possible solution for each exercise will be provided at the end of the session

High-level patterns

- Here we consider *ParallelFor ParallelForReduce* as data-parallel patterns
- *Macro-Data-flow* (MDF) as data-flow pattern (or task-parallel pattern)
- *Pipeline* and *task-farm* are high-level patterns as well !
- Other patterns available in FastFlow are:
  - *PoolEvolution* for modelling evolutionary applications
  - *Stencil2D* and *StencilReduce* patterns for iterative stencil-like computation (multi-core and CUDA-based GPGPUs)
  - *Divide&Conquer* (preliminary version)
  - *oclMap* and *cudaMap* patterns

High-level patterns: ParallelFor

map pattern

```
// sequential code
for(long i=0;i<N; i+=2)
  A[i] = f(i);
```

```
// parallel code
ParallelFor pf;
pf.parallel_for(0, N, 2, [&A](const long i) {
    A[i] = f(i);
});
```

- Loops with independent iterations may be parallelised using the ParalleFor pattern
- The ParallelFor interface is in the *parallel_for.hpp* file
- It is implemented on top of the task-farm with a suitable scheduling strategy
- There are many different methods that can be used
- Iteration scheduling provided:
  - Default static scheduling
  - Static scheduling with interleaving by using **parallel_for_static**
  - Dynamic scheduling
- Also provides *active scheduling* (by using farm's Emitter) and *passive scheduling*

High-level patterns: ParallelForReduce

map-reduce pattern

```
// sequential code: summing all elements
// of an array
double sum=0.0;
for(long i=0;i<N; i++)
  sum += A[i];
```

```
// parallel code
ParallelForReduce<double> pfr;
pf.parallel_for_reduce(sum, 0.0, 0,N,
         [&A](const long i, double &sum) {
            sum +=A[i];
}, [](double &sum, const double v) {
     sum+=v;
   }
);
```

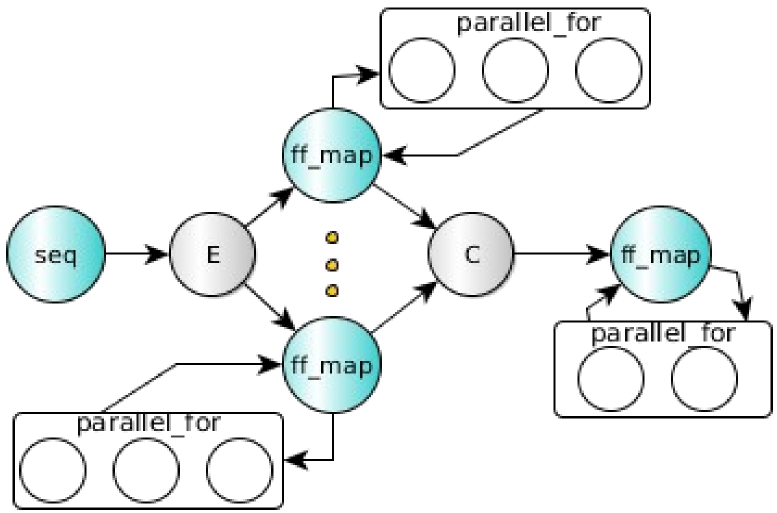
- A ParallelFor plus a reduction operation
  - associative and commutative operation
- The ParallelForReduce interface is in the *parallel_for.hpp* file
- It is implemented on top of the task-farm with a suitable scheduling strategy
- Executes a local reduction in the body part using a private variable plus a final reduction operation using a combination function.
- Scheduling strategies are the same as those provided by the ParallelFor pattern

Simple tests using a ParallelFor

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

High-level patterns: ff_map

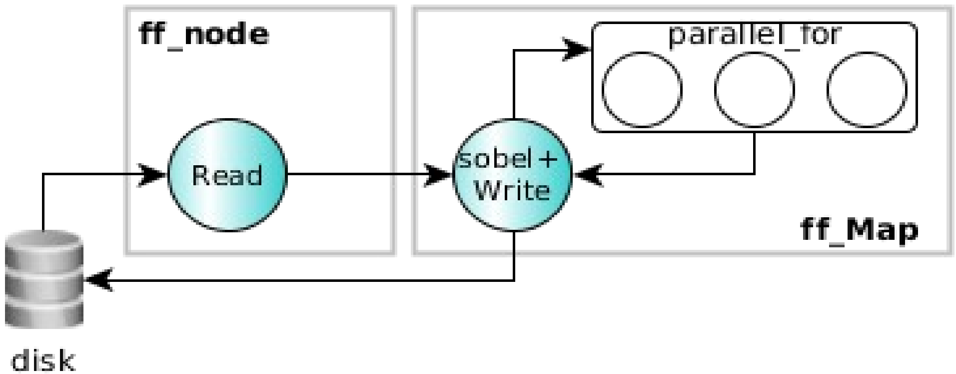
map pattern inside stream parallel patterns




```
struct mapWorker: ff_Map<> {
 void *svc(void *task) {
   ....
  ff_Map<>::parallel_for(...);
  ff_Map<>::parallel_reduce(....);
  return task;
 }
};
```

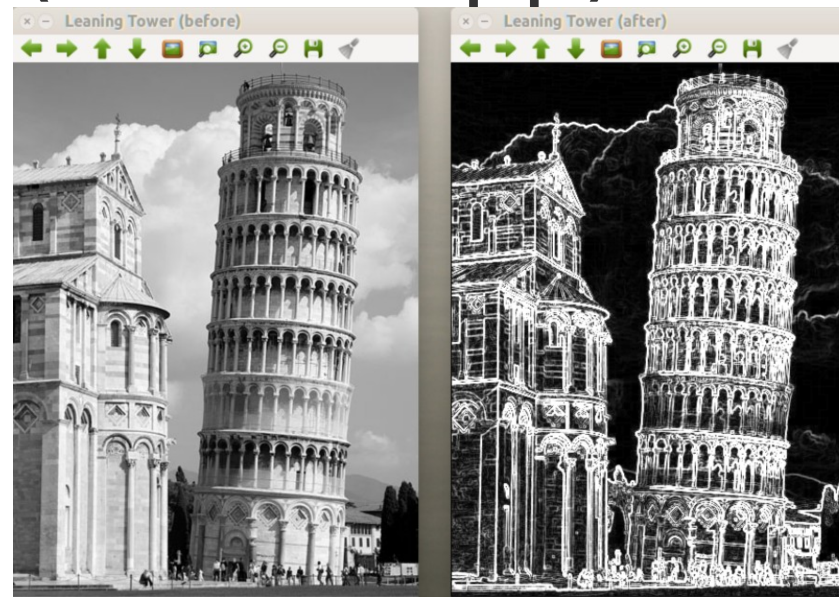
- The *ff_map* is just an *ff_node* that wraps a ParallelForReduce pattern
- The ff_map can be used as a pipeline stage and as a farm worker
- It is better to use the ff_map than a plain ParallelFor in a pipeline or farm computations because the run-time knows that the given stage/worker is parallel
  - Better thread mapping strategies and optimizations can be applied

Examples: Sobel filter (ffsobel.cpp)




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

  void *svc(void *t) {
    Task *task=reinterpret_cast<Task*>(t);
    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

Let's see the code!

Proposed exercises using ParallelFor & ParallelForReduce

- Simple matrix computation. Given in input a square matrix of size N compute the resulting value as:

$$\sum_{i=0}^{N-1} A[i][i] + \sum_{j=i+1}^{N-1} A[i][j] * A[j][i]$$

For example, given the following 3x3 matrix, then:

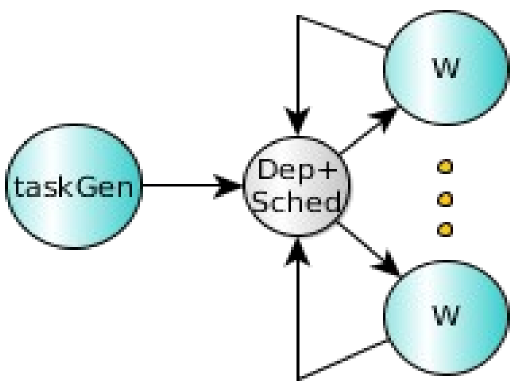
1	2	3
4	5	6
7	8	9

result = 1 + 2*4 + 3*7 + 5 + 6*8 + 9 = 92

  - The sequential implementation is given in ***matcomp.cpp***
  - The objective is to implement the computation in parallel using the ParallelForReduce pattern.

High-level patterns: ff_mdf

data-dependency pattern




```
void taskGen(ff_mdf*const mdf) {
   ….
   const param_info _1=  {&A, INPUT};
   const param_info _2 = {&B, INPUT};
   const param_info _3 = {&C, OUTPUT};
   std::vector<param_info> Param = {_1,_2,_3};

   mdf->AddTask(Param, GEMM, A,B,C);
   ….
}

ff_mdf mdf(taskGen, ...);
mdf.run_and_wait_end();
```

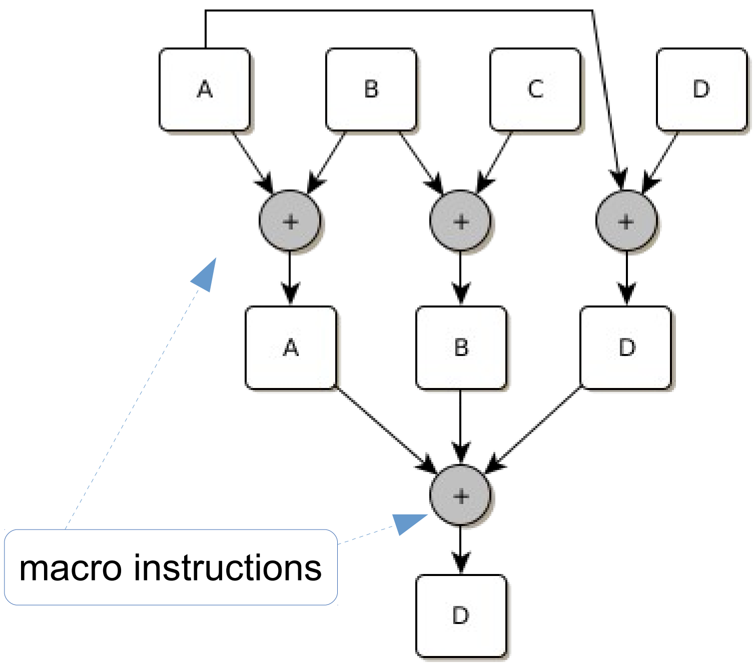
- The ff_mdf pattern targets ***macro-data-flow*** computations
- *Is a general approach to parallelism based only on data dependencies.*
- The computation is expressed by the data-flow graph, i.e. DAG whose nodes are macro-instructions and arcs are pure data-dependencies
  - A macro-instruction can be a set of simple instractions or a complex kernel function.
- By using the ff_mdf pattern, the user has to specify data-dependencies, i.e. declaring which are INPUT and OUTPUT data
- The **AddTask** method of the ff_mdf class is used to generate tasks
- The run-time, *automatically*, takes care of dependencies and then schedules ready tasks to Workers which executes ready (*fireable*) instructions in parallel

A simple test using ff_mdf

```
// macro operations
void sum2(long *X, long *Y, long size);
void sum3(long *X, long *Y, long *Z, long size);
....
// task generator function
void taskGen(Parameters<ff_mdf> *P) {
  ... auto mdf = P->mdf;

  {// A= A+B
   const param_info _1=  {&A, INPUT};
   const param_info _2 = {&B, INPUT};
   const param_info _3 = {&A, OUTPUT};
   std::vector<param_info> Param = {_1,_2,_3};
   mdf->AddTask(Param, sum2, A, B, SIZE);
  }{// B= B+C
   const param_info _1=  {&B, INPUT};
   const param_info _2 = {&C, INPUT};
   const param_info _3 = {&B, OUTPUT};
   std::vector<param_info> Param = {_1,_2,_3};
   mdf->AddTask(Param, sum2, B, C, SIZE);
  ......
```

A,B,C,D are arrays of size N




```
Parameters<ff_mdf> P;// structure containing all parameters needed to taskGen function
ff_mdf dag(taskGen, &P);  // creates the mdf object
P.A=A,P.B=B,P.C=C....P.mdf=&dag...   // preparing all parameters
dag.run_and_wait_end();  // run and wait termination
```

Simple test using the ff_mdf

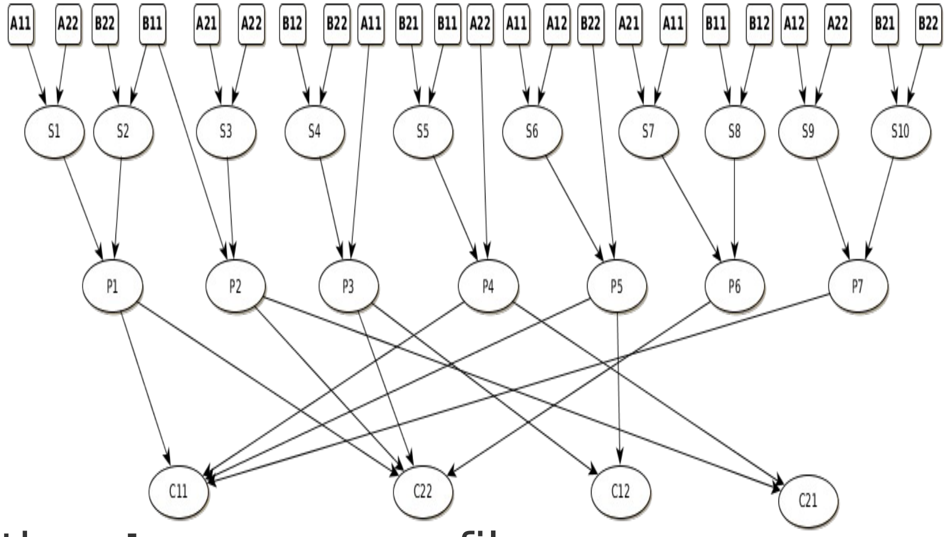
- Let's comment the code of the simple test presented in the FastFlow tutorial:
  - wf.cpp

Proposed exercise using ff_mdf

- Matrix multiplication using Strassen's algorithm:
  - We want to compute AxB = C, A is M by P, B is P by N and C is M by N
  - Partitioning the matrices in 4 equal-size blocks we have:

$$\begin{array}{|c|c|}\hline A11 & A12 \\ \hline A21 & A22 \\ \hline \end{array} \ \mathrm{X} \ \begin{array}{|c|c|}\hline B11 & B12 \\ \hline B21 & B22 \\ \hline \end{array} = \begin{array}{|c|c|}\hline C11 & C12 \\ \hline C21 & C22 \\ \hline \end{array}$$

S1 = A11 + A22   S2 = B11 + B22   P1 = S1 * S2
S3 = A21 + A22   P2 = S3 * B11
S4 = B12 – B22   P3 = A11 * S4
S5 = B21 – B11   P4 = A22 * S5
S6 = A11 + A12   P5 = S6 * B22
S7 = A21 – A11   S8 = B11 + B12   P6 = S7 * S8
S9 = A12 – A22   S10 = B21 + B22   P7 = S9*S10
C11 = P1 + P4 - P5 + P7
C12 = P3 + P5
C21 = P2 + P4
C22 = P1 - P2 + P3 + P6



  - The sequential code is provided in the **strassen.cpp** file
- Write a parallel version using the ff_mdf pattern.

Thanks for participating!

For any questions or comments please send an e-mail to torquati@di.unipi.it