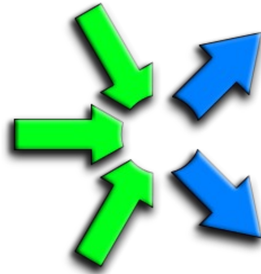
GridKa School 2015

Big Data Virtualization Modern Programming

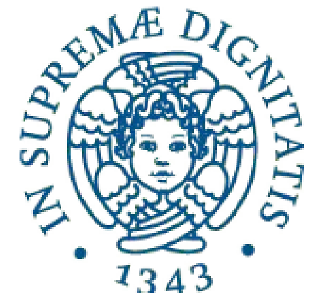


Introduction to FastFlow programming

Hands-on session

Massimo Torquati <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy



- FastFlow basic concepts
  - skeletons and patterns
  - stream concept
  - building blocks
- Core patterns:
  - *pipeline* & *task-farm*
- High-level patterns:
  - ParallelFor* and Map
  - Macro-DataFlow (mdf)
  - OpenCL StencilReduceLoop
- Targeting distributed system (basic concepts)

The FastFlow tutorial

- Ask for the VM password in case you want to use the Linux VM provided by GridKa
- Update the FastFlow version contained in the VM
  - cd fastflow
  - svn update
- The FastFlow tutorial is available as pdf file on the GridKa wiki page
  - "FastFlow tutorial"
    - http://wiki.scc.kit.edu/gridkaschool/index.php/Main_Page
- 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...

Objectives

- Understand the Structured Parallel Programming methodology
- 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 FastFlow programs

Structured parallel programming

- Structured parallel programming aims to provide *standard* (and *effective*) rules for composing parallel computations in a machine-independent way
  - **Goal**: *reducing the complexity of parallelization problems by introducing constraints*

    i.e. restricting the computation structure
  - *Modularity portability* and programmability are the keywords
- *Parallel paradigms* are the base components of parallel applications
- Using structured parallel programming force to think parallel
- The programmer is relieved from all concerns related to the implementation, he/she concentrates "*only*" on computational aspects

Separation of concerns principles

Skeletons & Patterns

Algorithmic Skeletons

- From HPC community
- From early '90
- Pre-defined parallel high-order functions provided as constructs or lib calls

Parallel Design Patterns

- From SW engineering community
- From early '00
- “Recipes“ to handle parallelism (name, problem, algorithms, solutions, ...)

- *The same concept at different abstraction levels*
- We use the two terms patterns and skeletons, interchangeably.
  - We want to emphasise the similarities of these two concepts

Using patterns

Avalilable parallel patterns

Map, Reduce, Stencil,....

Pipeline, Task-Farm

Divide&Conquer, MDF, ….

Data Parallel

Stream Parallel

Task Parallel

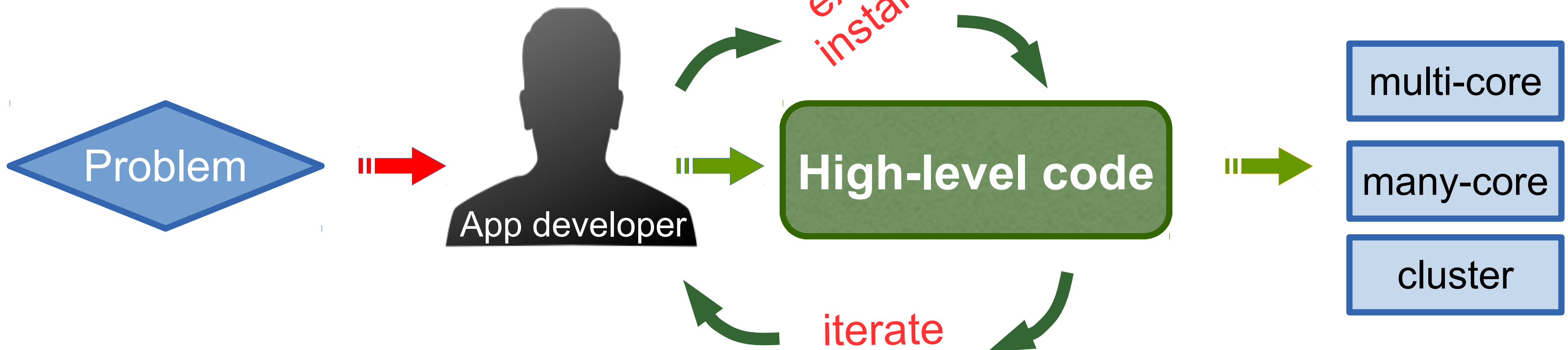
System developer

run-time support

Which pattern ?

select
compose
extend
instantiate

Target platforms



Assessment

Separation of concerns

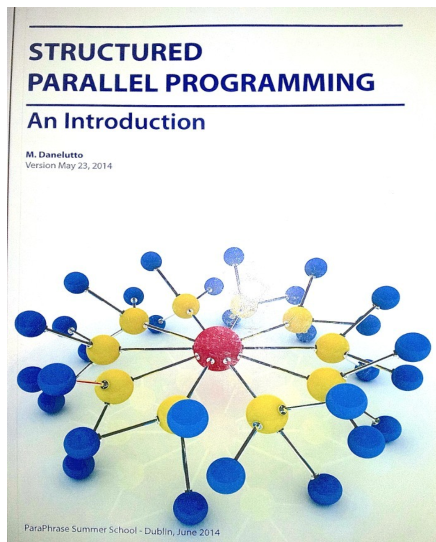
- Application programmer: what is computed
- System programmer: how the result is computed

Inversion of control

- Program structure suggested by the programmer
- The run-time selects the optimization for the target platform

Performance

- Close to hand tuned code (sometimes better)
- Reduced development time. Lower total cost to solution.



"Structured Parallel Programming" by Marco Danelutto

Available on-line as SPM course material at M. Danelutto web page

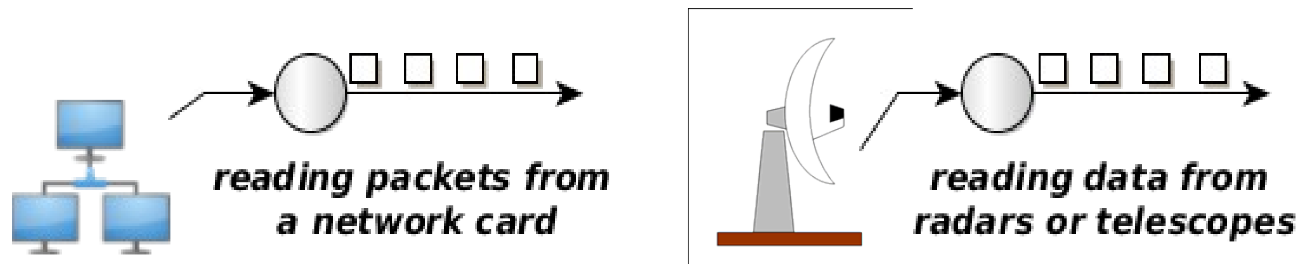
http://www.di.unipi.it/~marcod

Stream concept

- The stream concept is important in FastFlow
- A stream is a sequence of values (possibly infinite), coming from a source
  - 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

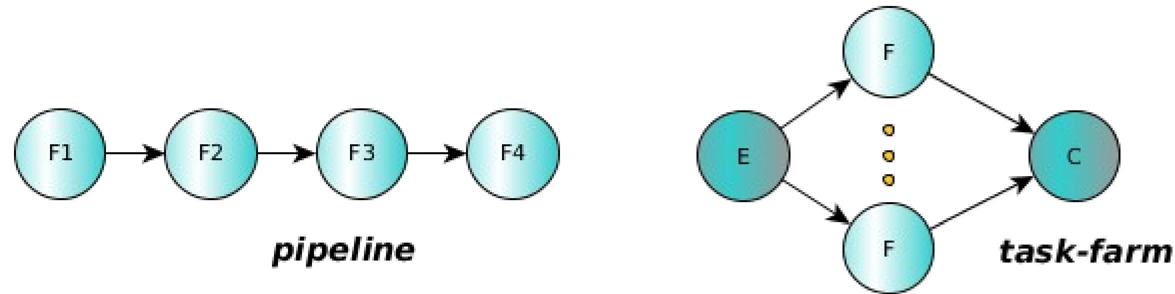
- "*real streams*"



  - 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

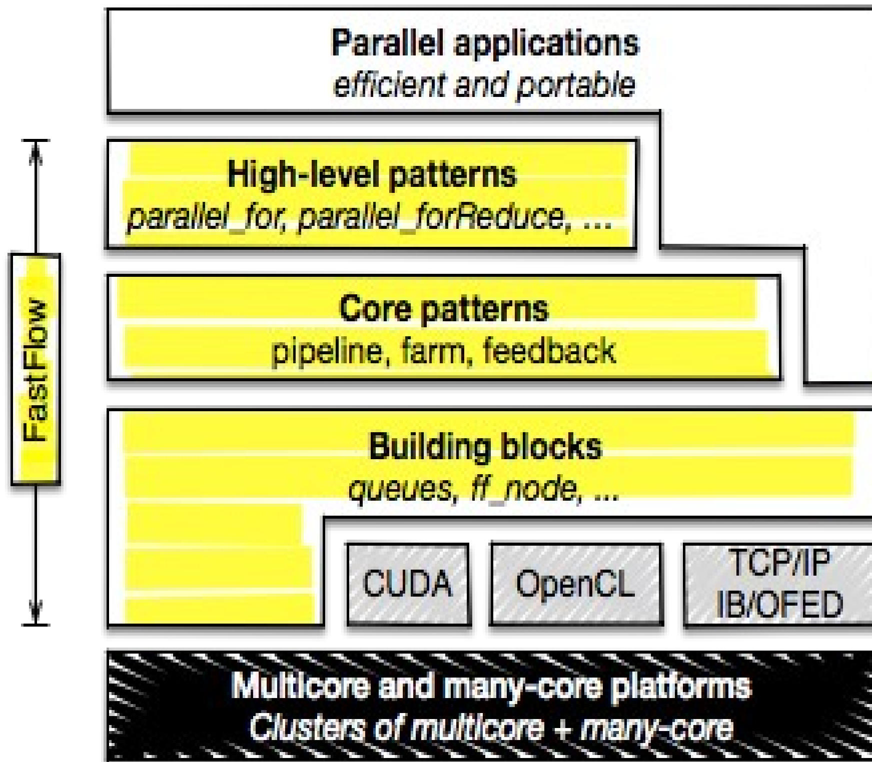
```
for(i=start; i<stop; i+=step)
   allocate data for a task
   create a task
   send out the task
```

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

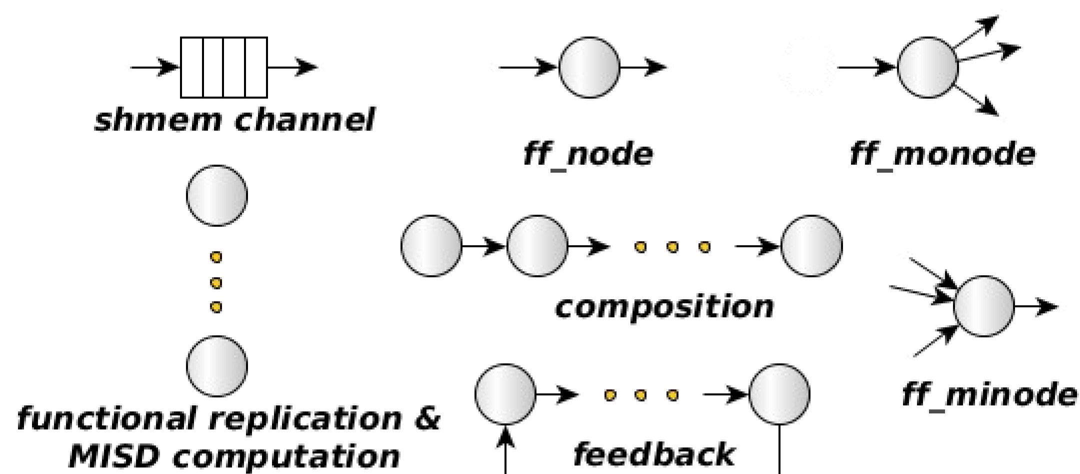
The FastFlow layers



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

- 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

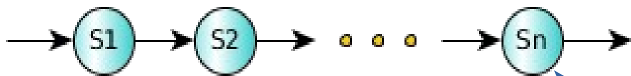
Building blocks



- 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 Parallel Patterns ("core" patterns)

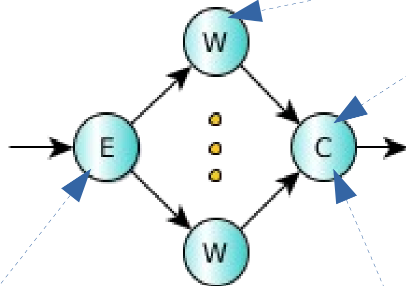
pipeline




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

ff_node

task-farm



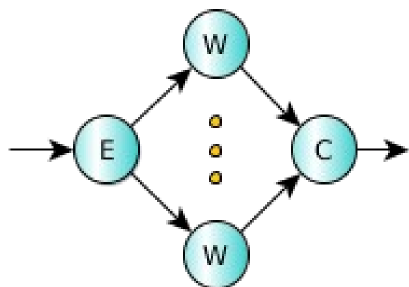

```
std::vector<std::unique_ptr<ff_node> >  W;
myEmitter E;
myCollector C;

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

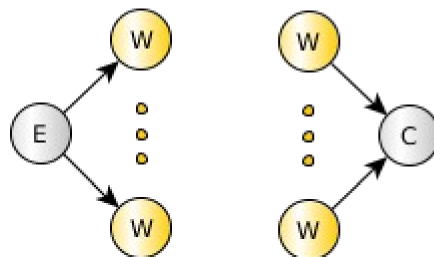
Emitter:
schedules input data items

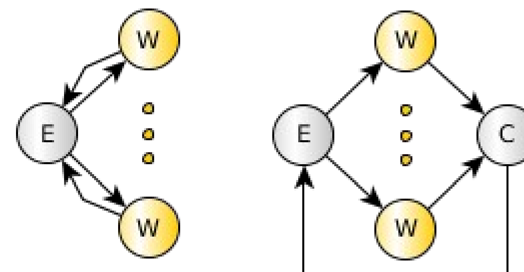
Collector:
gathers results

Stream Parallel Patterns ("core" patterns)

task-farm





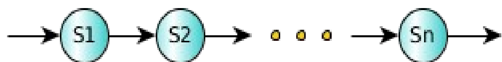


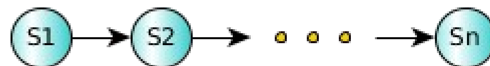
task-farm

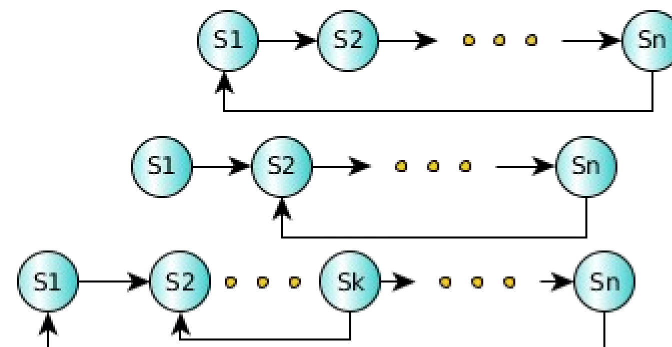
task-farm + feedback

Specializations

pipeline





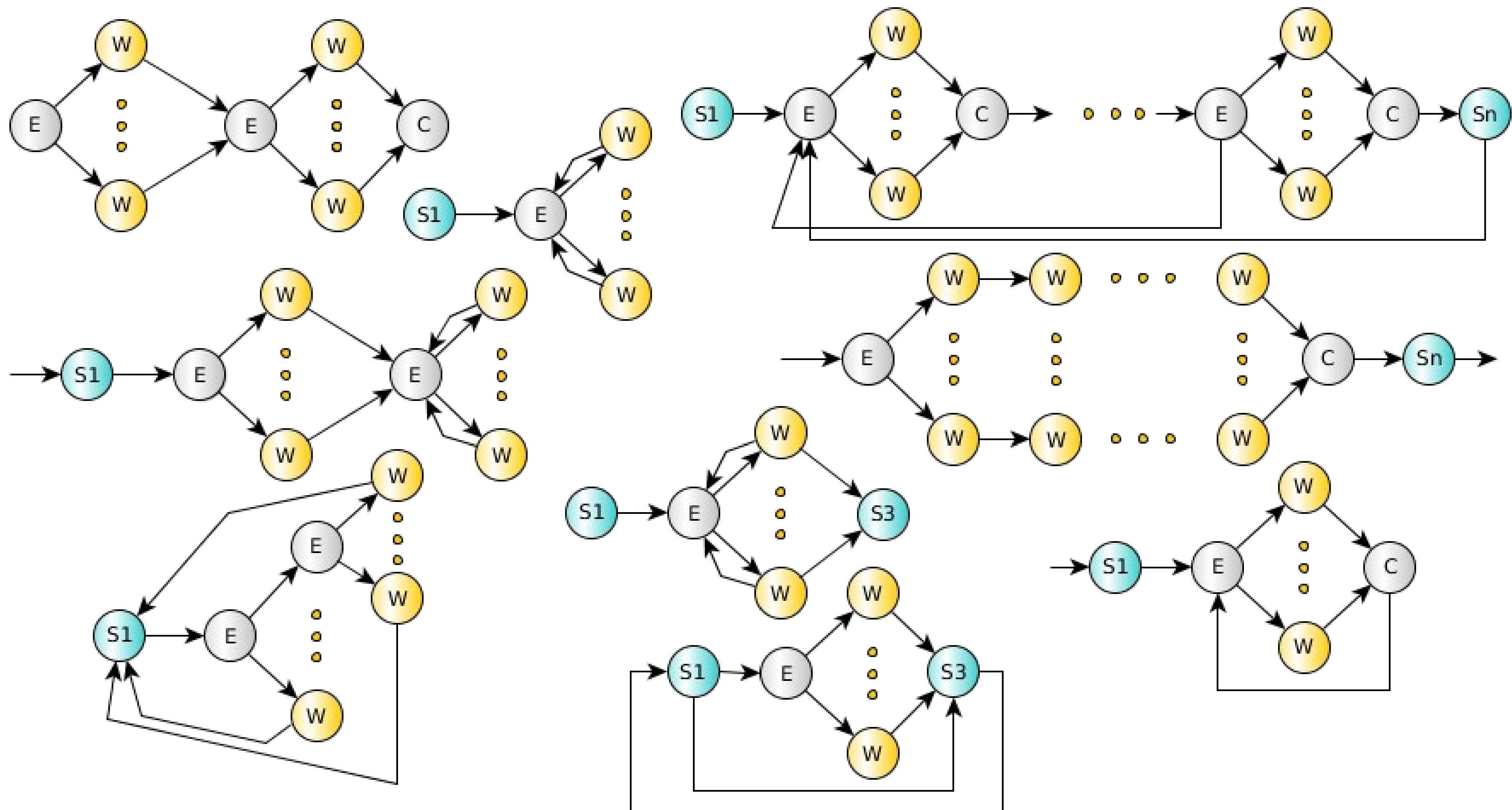


pipeline

pipeline + feedback

Patterns

Core patterns composition



pipeline + task-farm + feedback

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(void *task) {
        return task;
}};
struct myNode3: ff_node_t<myTask> {
  myTask *svc(void * 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 comment on the code of the 2 simple tests presented in the FastFlow tutorial:
  - hello_pipe.cpp
  - hello_pipe2.cpp

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);
```

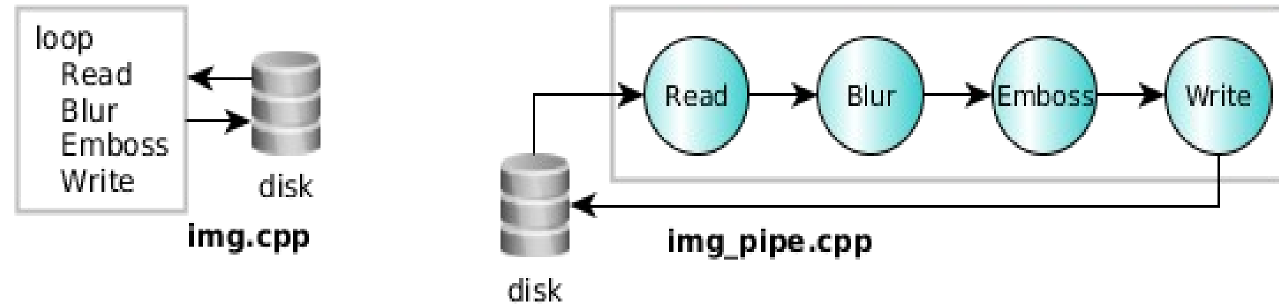
- Ordered task-farm pattern
- Tasks are produced in output in the same order as they arrive in input
- In this case it is not possible to redefine the scheduling policy

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

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




```
// 4-stage pipeline
ff_Pipe<Task> pipe( read, blur, emboss, write );
pipe.run_and_wait_end();
```

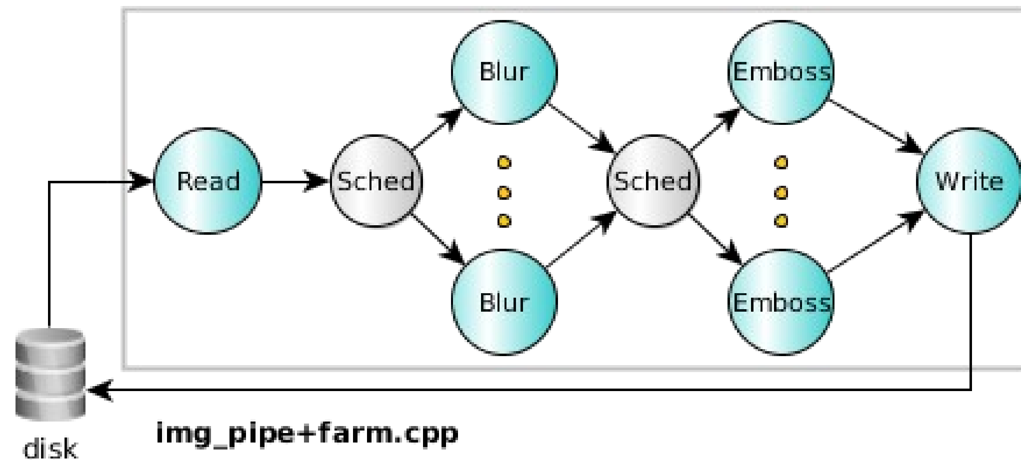
```
// 1st stage
struct Read: ff_node_t<Task> {
  Task *svc(Task *) {
     for(long i=0;i<num_images;++)
        Image *img = new Image;
        Img->read(filename);
        Task *task = new Task(img,filename);
        ff_send_out(task);
  }
  return EOS; // End-Of-Stream
};
```

```
// 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 reinterpret_cast<Task*>(GO_ON);
}
```

Examples: image filtering (img_pipe+farm.cpp)



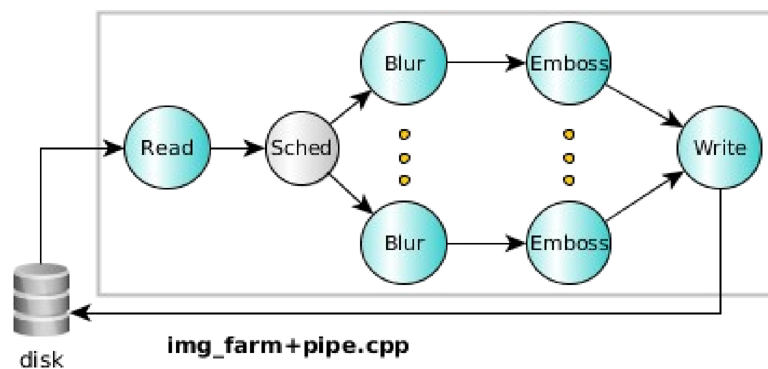

```
ff_Farm<Task> farmBlur(BlurFilter, numBlurWorkers);
farmBlur.remove_collector();
ff_Farm<Task> farmEmboss(EmbossFilter, numEmbosWorkers);
// 4-stage pipeline
ff_Pipe<Task> pipe( read, farmBlur, farmEmboss, write );
pipe.run_and_wait_end();
```

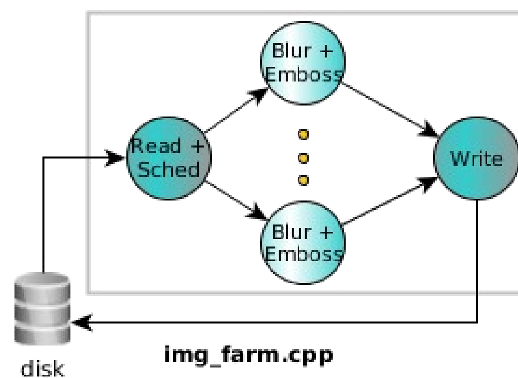
```
// ff_node wrapper to the Write function
struct Writer: ff_minode_t<Task> {
 Task *svc(Task *task) {
    return  Write(task, this);
};
```

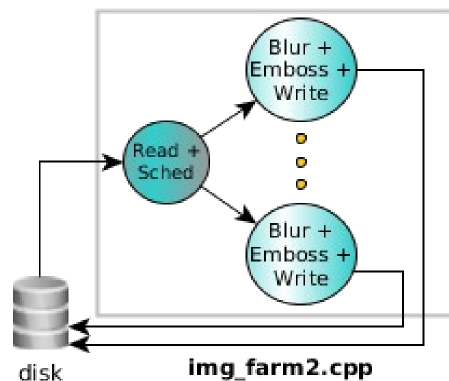
Other nodes are the same as before

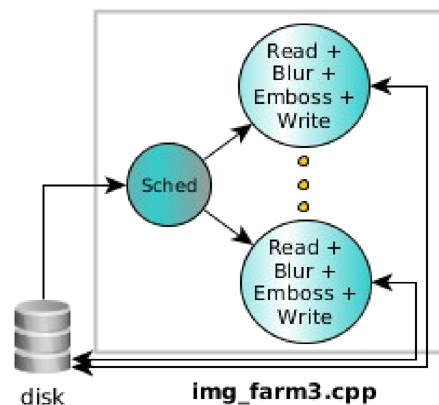
Examples: image filtering

Other simple transformations









Take a look at 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 first implement a pipeline version (read → compute → write), then transform the sequential compute stage in a task-farm.
    - simplecomp_pipe.cpp
    - simplecomp_farm.cpp
  - **HINT**: the structure is quite similar to the one used in the img_pipe.cpp and img_farm.cpp, respectively.

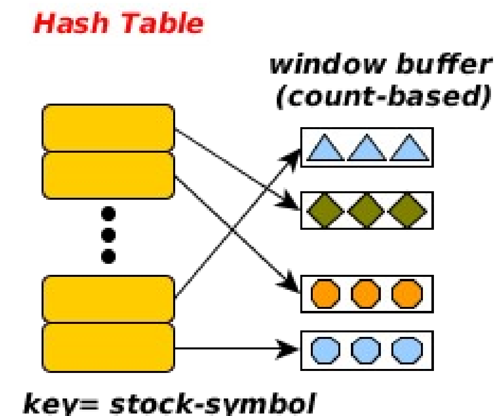
- A more complex and efficient implementation is left as homework
  - We will discuss together some possible solutions

Example: statefull pipeline

- Simplified financial application

Sequential pseudo-code:

```
Receiver rec(port); // recv quotes from the market
while( recv.receive(quote) ) {
  filterQuote(quote);  // filters data
  If ((wid=winManagement(quote, win_size, win_slide)) != -1) {
     computeWindow(wid, result);  // data ready
     writeOnDisk(result);  // write result
   }
}
```



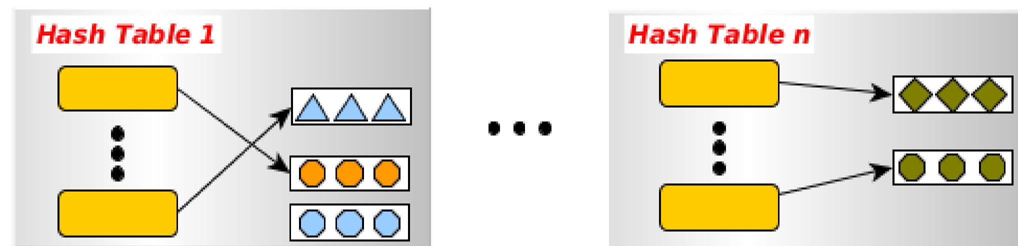
- WindowManagement is based on a hash-table containing different buffers for each stock symbol
- The application is logically a 3-stage pipeline (receive, compute, write) but the middle stage cannot be replicated …. unless ….

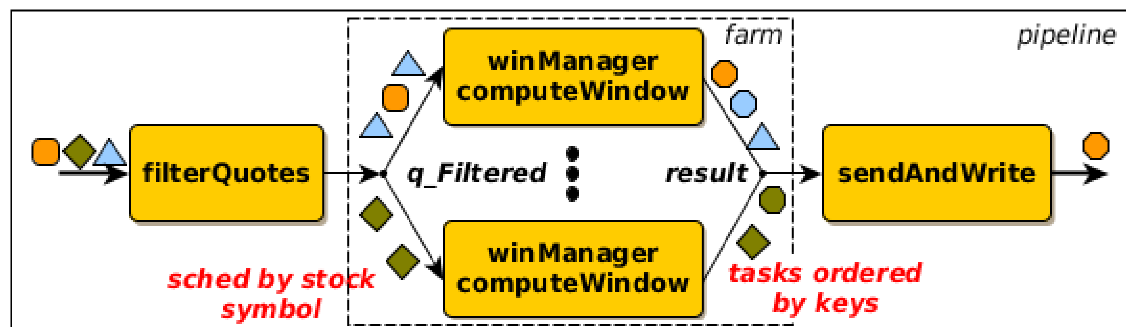
Example: statefull pipeline

- … unless … **the hast-table is *partitioned*** among all workers and the quotes are scheduled by stock symbol

Parallel structure:

```
struct firstStage: ff_node_t<quote_t> {
  quote_t *svc(quote_t *in) { return filter(*in); }
};
….
Receiver rec(port);
firstStage  first(rec);

std::vector<std::unique_ptr<ff_node>> W;
for(long i=0;i<nworkers;++i)
     W.push_back(make_unique<compute>
                         (win_size,win_slide));
ff_Farm<task_t,ret_t> farm(std::move(W));
Scheduler<decltype<SchedF> Sched(schedF);
farm.add_emitter(Sched);
farm.remove_collector();

lastStage  last(writerOnDisk);

ff_Pipe<> pipe(first, farm, last);
pipe.run_and_wait_end();
```





Potential load balancing problems !

High-level patterns

- Here we consider
  - *ParallelFor** data-parallel patterns
    - ff_Map, that can be used in pipeline and task-farm
  - *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
  - *StencilReduceLoop* patterns for iterative stencil-like computation (multi-core, CUDA/OpenCL-based GPUs)
    - We will introduce some basic concept later

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

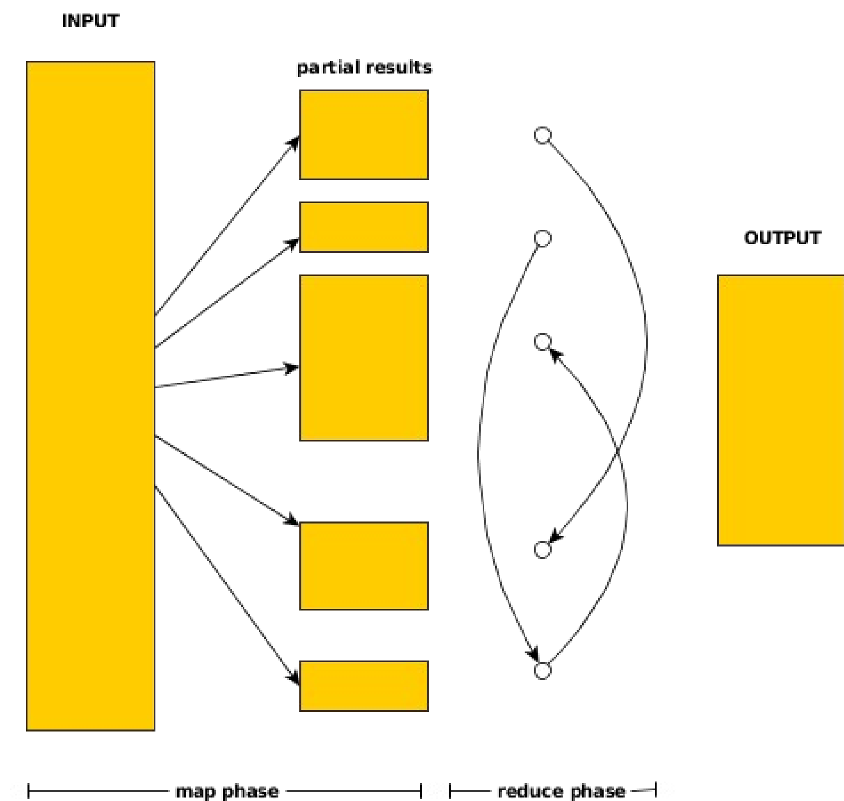
```
// sequential code: summing all elements
// of an array
double sum=0.0;
for(long i=0;i<N; i++)
  sum += A[i];
```

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

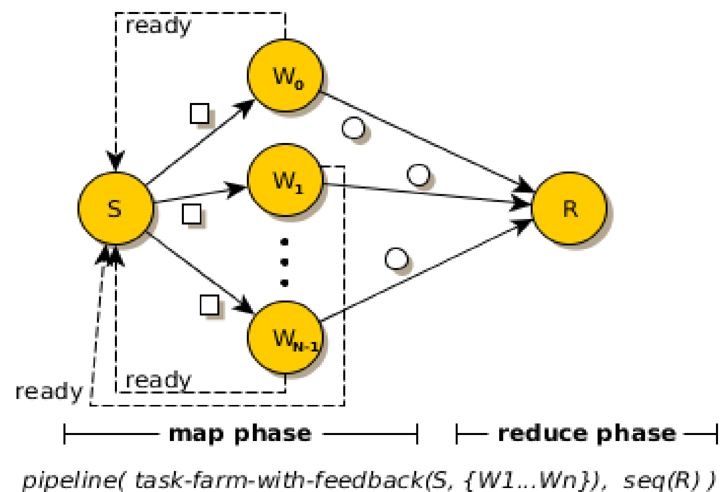
- A ParallelFor plus a reduction operation
  - associative operation
- The ParallelForReduce interface is in the *parallel_for.hpp* file
- It is implemented on top of the task-farm with a suitable scheduling strategy
- Executes a local reduction in the body part using a private variable plus a final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

High-level patterns: ParallelForPipeReduce

map-reduce pattern



- Useful when the reduction function has to be executed serially
- Offers the option to compute the Map part in pipeline with the Reduce part.
- The ParallelForPipeReduce interface is in the *parallel_for.hpp* file
- Scheduling strategies are the same as those provided by the ParallelFor pattern

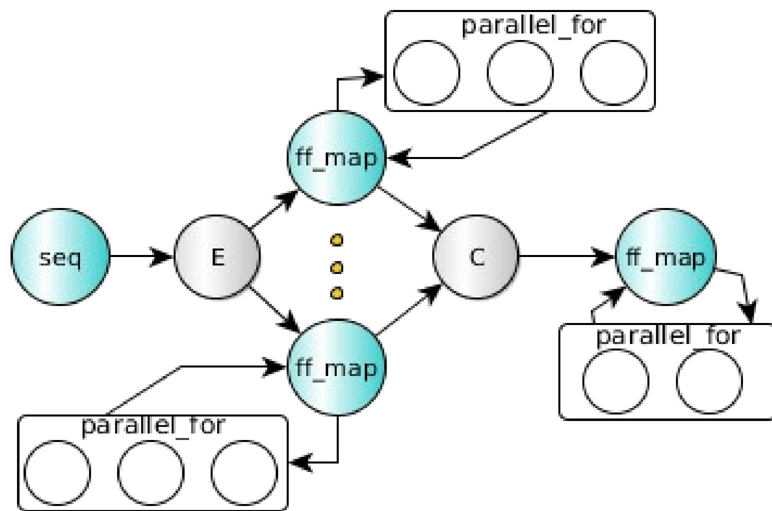


Simple tests using a ParallelFor

- 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

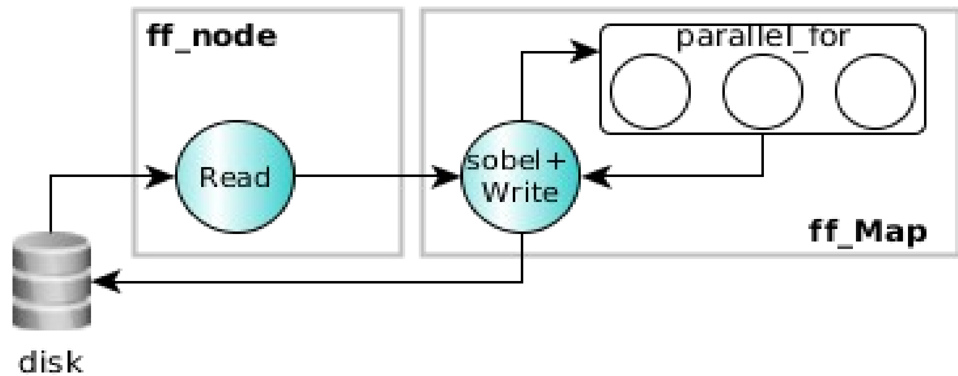
map pattern inside stream parallel patterns



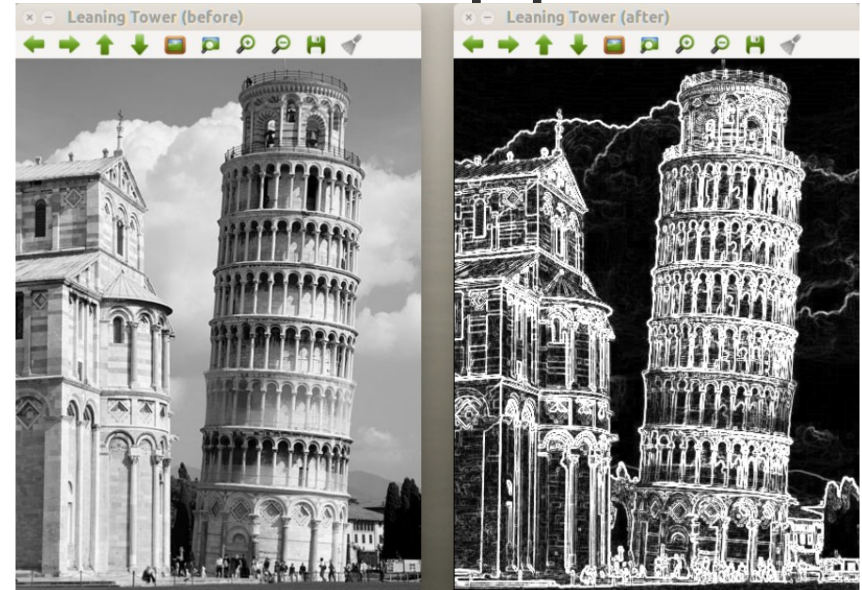

```
struct mapWorker: ff_Map<> {
 void *svc(void *task) {
   ....
  ff_Map<>::parallel_for(...);
  ff_Map<>::parallel_reduce(....);
  return task;
 }
};
```

- 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<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);
   }
```

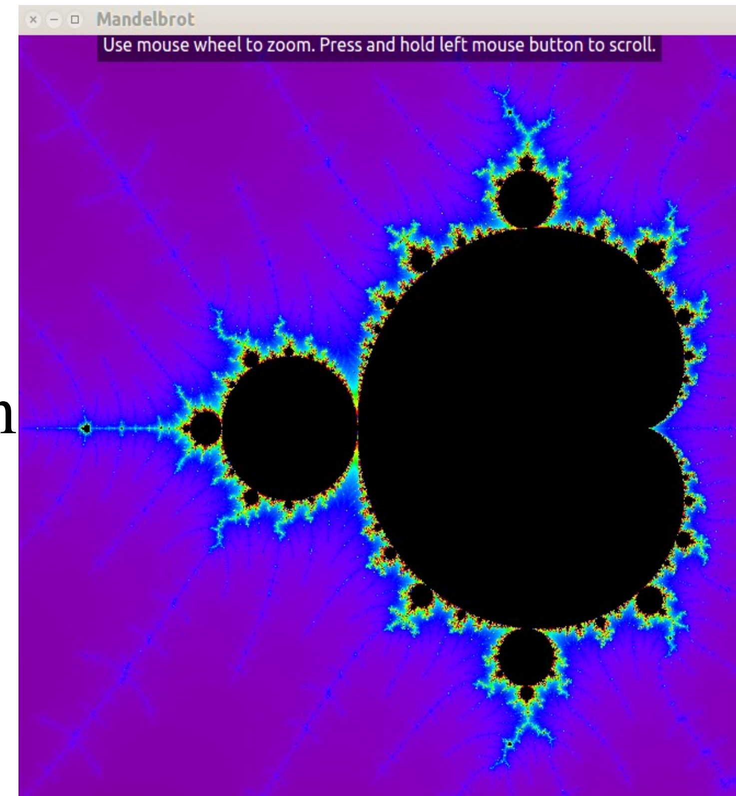


- The first stage reads a number of images from disk one by one, converts the images in B&W and produces a stream of images for the second stage
- The second stage applies the Sobel filter to each input image and then writes the output image into a separate disk directory

Let's see the code!

Examples: Mandelbrot set

- 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

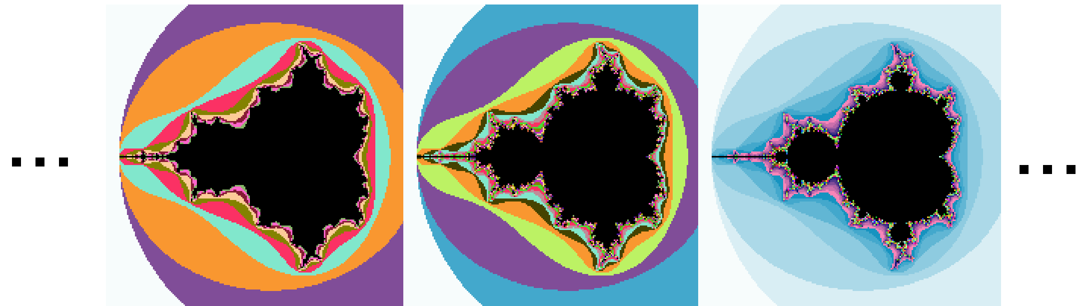


  - Let's consider the minimum computation unit a single image line (image size 2048x2048, max 10^3 iterations per point)
    - ParallelFor Static partitioning of lines (48 workers) **MaxSpedup 14**
    - ParallelFor Dynamic partitioning of lines (48 workers) **MaxSpeedup 37**

Data-partitioning may have a big impact on the performance

Examples: Mandelbrot set (2)

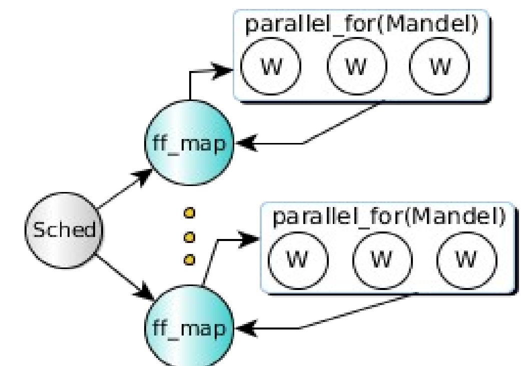
- Suppose now we want to compute a number of Mandelbrot images (for example varying the computing threshold per point)



- 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

Proposed exercises using ParallelFor & ParallelForReduce

- Finding the minimum and the index of the minimum value in an array of integer values of size N.

  Given the following array:

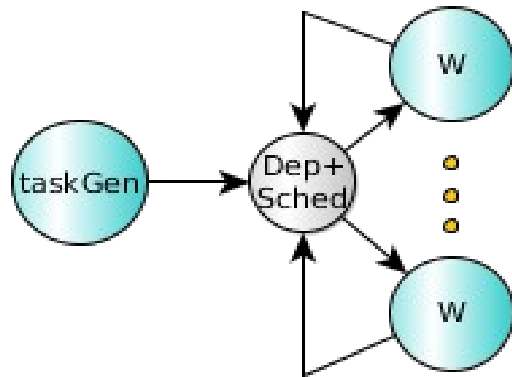
0	1	2	3	4	5	6	7	8	9
31	52	11	13	3	12	23	64	2	12

  we want as output the pair <8, 2>

  - The sequential implementation is given in **arrayminindex.*cpp***
  - The objective is to implement the computation in parallel using the ParallelFor* patterns.

High-level patterns: ff_mdf

data-dependency pattern



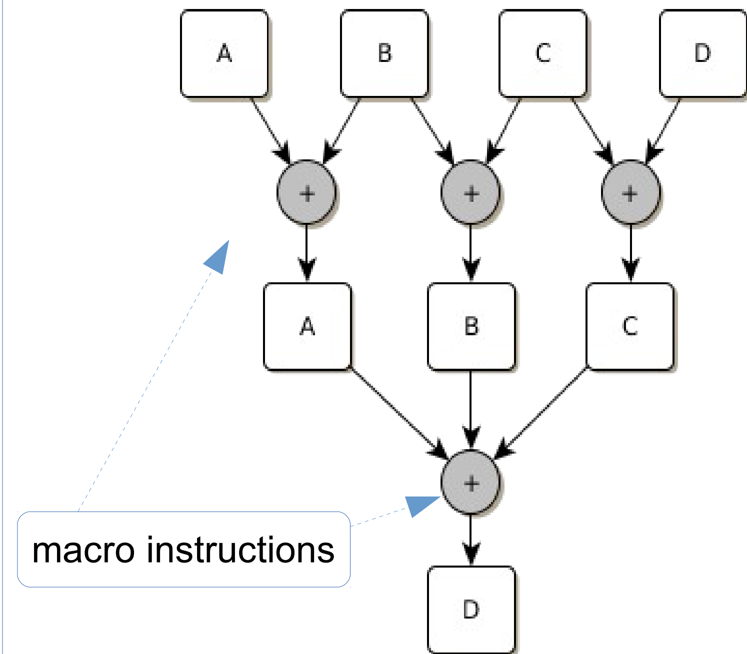

```
void taskGen(ff_mdf*const mdf) {
   ….
   const param_info _1=  {&A, INPUT};
   const param_info _2 = {&B, INPUT};
   const param_info _3 = {&C, OUTPUT};
   std::vector<param_info> Param = {_1,_2,_3};

   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



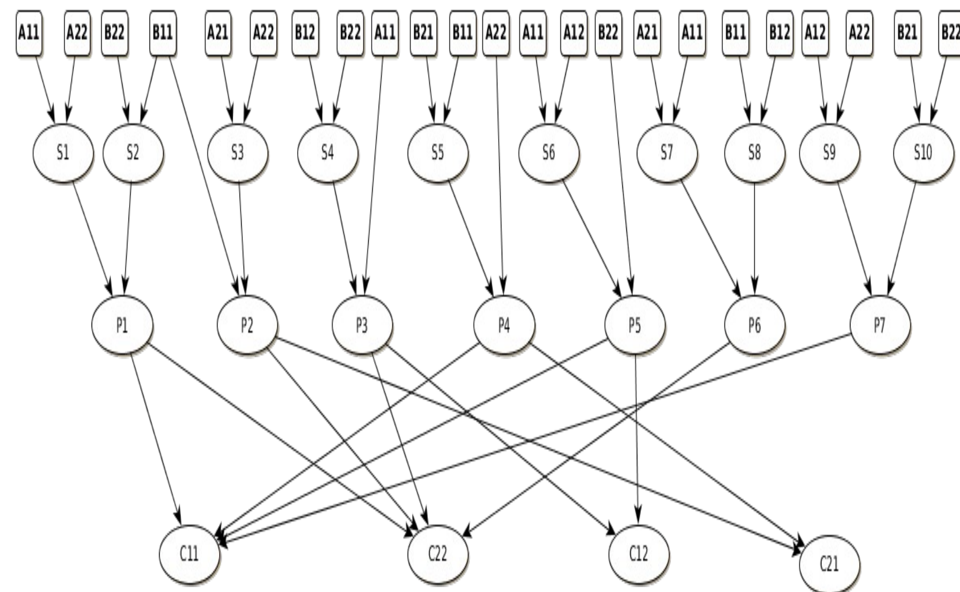

```
Parameters<ff_mdf> P;// structure containing all parameters needed to taskGen function
ff_mdf dag(taskGen, &P);  // creates the mdf object
P.A=A,P.B=B,P.C=C....P.mdf=&dag...   // preparing all parameters
dag.run_and_wait_end();  // run and wait termination
```

Strassen algorithm example using ff_mdf

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

A11	A12		B11	B12		C11	C12
A21	A22	X	B21	B22	=	C21	C22

```
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
```



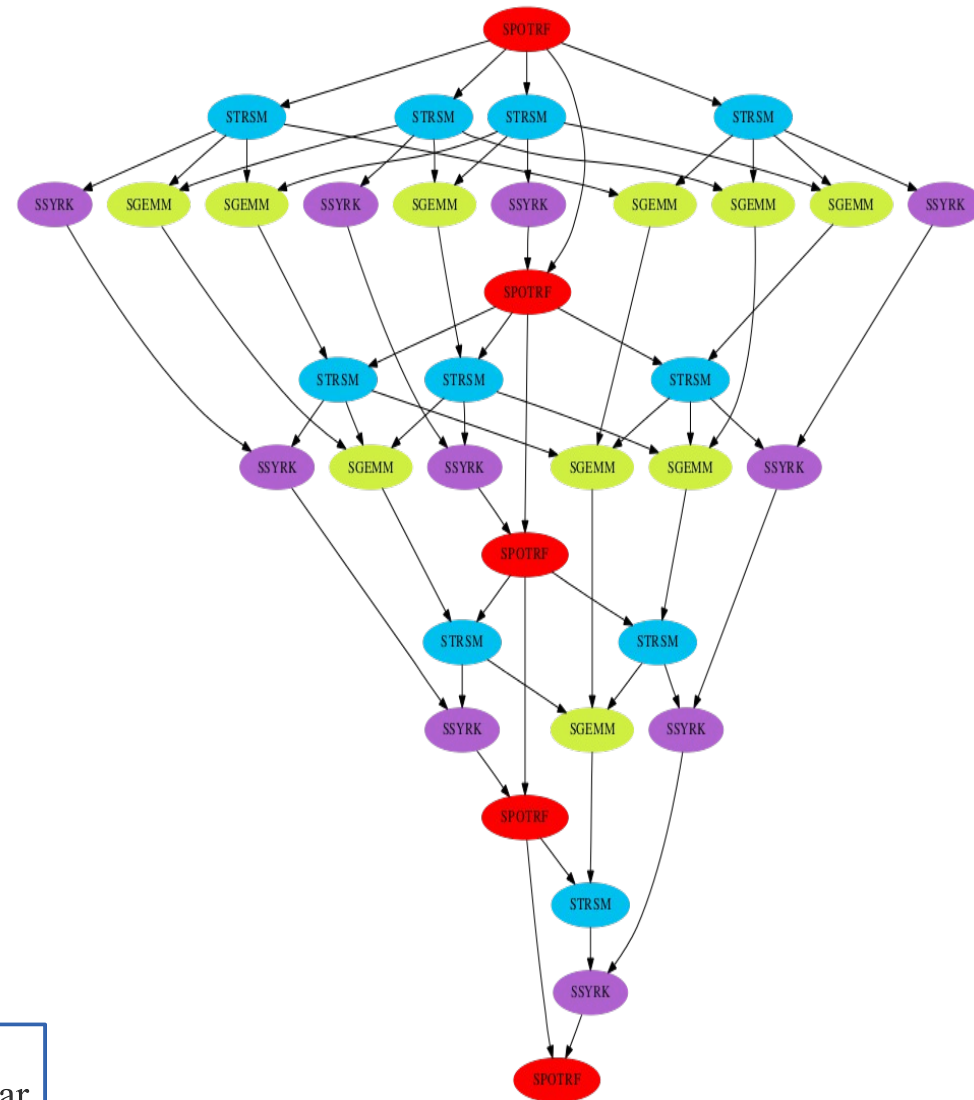
- Let's take a look at the code **examples/strassen/strassen_mdf.cpp** in the fastflow folder.

Block Cholesky and LU factorizations

- Much more complex data-flow graphs
- The DAG is huge for big matrices
  - cannot be entirely stored in memory
  - Have to be generated dynamically

- The *ff_mdf* pattern allows to generate the graph dynamically and overlap the graph generation phase with the computation

D. Buono, M. Danelutto, T. De Matteis, G. Mencagli and M. Torquati "A light-weight run-time support for fast dense linear algebra on multi-core" in PDCN 2014 conference, 2014



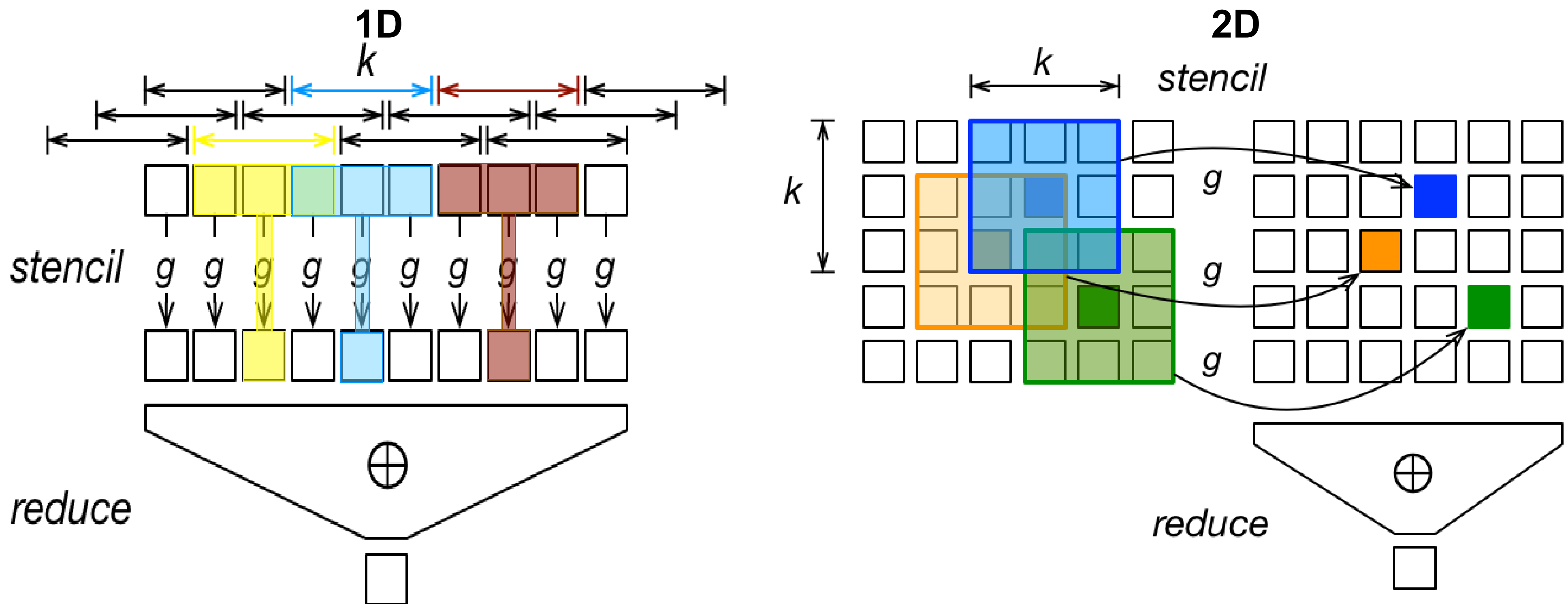
DAG represents, 5 tiles, left-looking version of Cholesky algorithm

(not yet covered in the FastFlow tutorial)

The Loop-Of-Stencil-Reduce pattern

- We introduced the **Loop-Of-Stencil-Reduce** pattern
  - Specifically targeting iterative Map+Reduce algorithm
  - Both 1D and 2D computation on multiple GPUs
  - On top of this pattern we implemented: map, reduce and map+reduce
  - OpenCL and CUDA version

Stencil computation



$$\textbf{stencil}\ g\ k\ a' = [g(S_0), g(S_1), \ldots g(S_{n-1})]$$

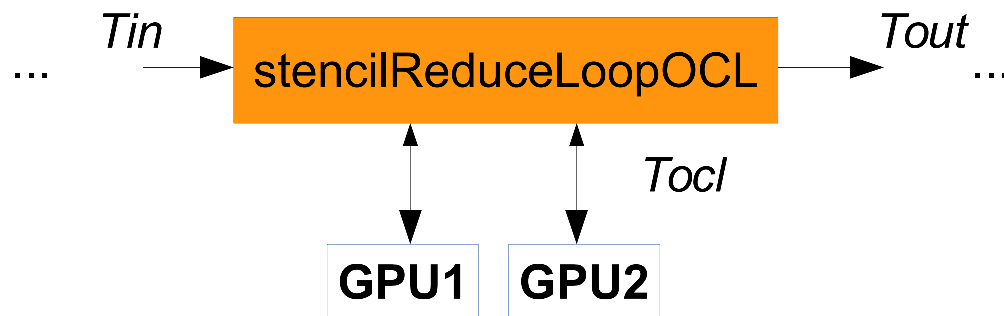
- g : list of T → list of T is the *elemental function*
- $S_i = [a'_{i-k}, \ \ldots, a'_i, \ \ldots, a'_{i+k}]$ is the i-*th neighborhood*
- *we consider the case where k is a constant value*
- *a' is the extension of a with suitable extra items to deal with the border neighborhood computation*

ff_stencilReduceLoop* run-time behaviour

```
// d2h input data, allocate double-buffering
(in_d_ptr, out_d_ptr, env_d_ptr) = allocate-write(input, env)
while cond
   //  swap buffers, swap halo among different devices
   swap_buffers_and_halo (in_d_ptr, out_d_ptr)
   out_d_ptr = stencil <f_kernel> (in_d_ptr, env_d_ptr) // DEV
   part_d_ptr = reduce <op_kernel> (out_d_ptr) // DEV
   partial_data = read(part_d_ptr)
   cond = reduce op partial_data // host final reduction
// h2d−copy output, release buffers
output = release-read (in_d_ptr, out_d_ptr, env_d_ptr)
```

ff_stencilReduceLoop*

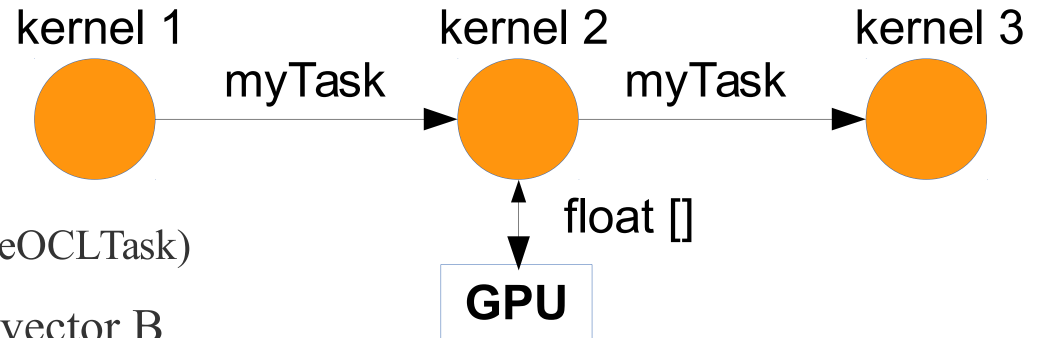
- Using OpenCL (for CUDA we have the same concepts)
- Implemented as a “special” kind of FastFlow node



- Each GPU is managed by a different internal “accelerator”
- Tocl is a template type encapsulating both input and output GPU types
  - It is used as an interface type via the setTask/releaseTask methods.
- Using the ff_stencilReduceLoopOCL we implemented:
  - ff_mapOCL, ff_reduceOCL, ff_mapReduceOCL
- Can be used as a pipeline stage or as a worker in a task-farm

Targeting GPUs in a pipeline

- Pipeline computation example:
  - Tin/Tout is myTask
  - Tocl is oclTask (must subclass from baseOCLTask)
- We want to compute a Map only on the vector B
  - B is non contiguous in memory




```
// this is the stream type (Tin/Tout)
struct myTask {

  std::vector<float> A;
  std::vector<std::vector<float> > B;
  ….
  std::string command;
  ….

};
```

```
// OpenCL interface type for the kernel2 node   (Tocl)
struct oclTask: baseOCLTask<myTask, float, float> {
   // called by the run-time for each input task
   void setTask(const Task *task) {
        const size_t size = computeSize(B);
        buffer = new float[size];
        copyBuffer(B, buffer, size);
        //set input and output host pointers
        setInPtr(buffer, size); setOutPtr(buffer);
   }
   // called by the run-time at the end of task execution
   void releaseTask(Task *task) {
        copyBack(task->B, buffer); // copy results back to B
        delete [] buffer;
    }
   float *buffer;
};
```

Examples: Sobel filter

- Let's take a quick look at the ff_pipe+mapOCL.cpp example in the fastflow/tests/ocl folder.

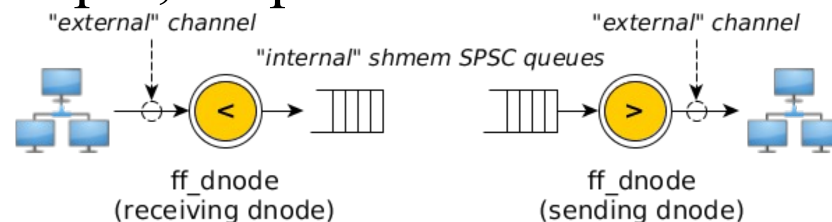
(not yet covered in the FastFlow tutorial)

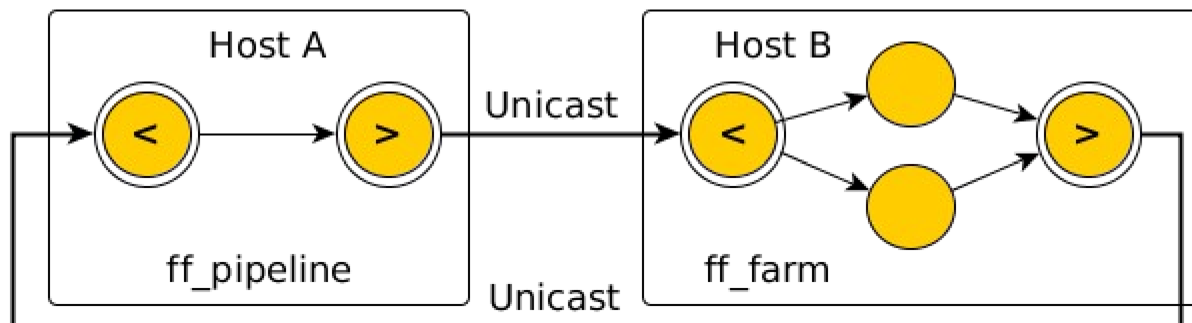
Targeting distributed systems (concepts)

- ***Building blocks:***
  - **ff_dnode** extend the ff_node class
  - **Communication patterns:**
    - *Unicast*: **one-to-one** unidirectional channel
    - *Broadcast*: **one-to-many**, the same data is sent to all connected peers
    - Scatter: **one-to-many**, data is split in disjoint partitions, each one sent to a distinct connected peer
    - On-demand: **one-to-one**, the data is sent to one of the connected peear using a request-reply protocol
    - Gather-All: **many-to-one**, receives a data partition from all connected peers, and eventually the message is recomposed
    - Collectc-from-Any: **one-to-one**, receives data from one of the connected peers
    - TODO: *Multicast*, ….
  -

ff_dnode

- A ff_dnode is an ff_node with an extra channel used to communicate with another dnode by using a communication pattern (external channel)
- The external channel may be specialized for input, output or both
- The idea is that ***only edge nodes*** are dnodes

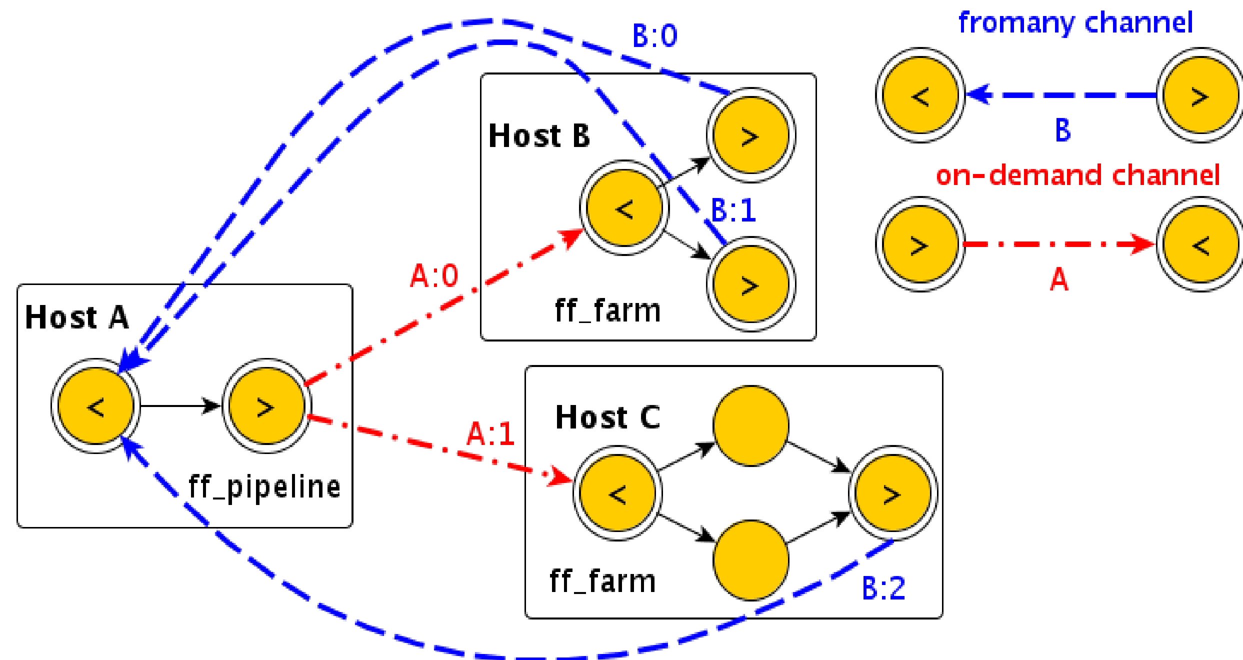




- Communication patterns:
  - TCP/IP networks by using the ZeroMQ library
  - Infiniband networks by using Linux verbs (experimental)

How it works

- The user has to:
  - Define and connect the application parts using proper communication patterns
  - Prepare the data to send
  - Provide the run-time with message buffers where data has to be received



How to define a dnode

emacs@pirotecnia

File Edit Options Buffers Tools C++ Help

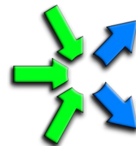
```
// just an edge-node of my application
class myNode: ff_dnode<zmqBcast> {
public:
  myNode(const std::string& name,          // unique identifier for the channel
         const std::string& address,       // peer address [host:port]
         const int npeers,                 // n. of peears
         zmqBcast::TransportImp* const t);// transport object

  int svc_init() {
    // I am the producer, i.e. the one that sends data out.
    // The consumer has to call init with the last param set to false.
    return ff_dnode<zmqBcast>::init(name,address,npeers,t, true);
  }

  void* svc(void* task) {
    // As soon as a task is returned or the ff_send_out method is called,
    // the message is sent out in broadcast (because of zmqBcast pattern)
    // to all connected peers.
  }

  void svc_end() { ....}
...
};
```

- Let's take a quick look at the img_d.cpp example which is a possible distributed version of the in the img.cpp app.
  - This version uses only 2 machines
  - It is just a proof-of-concept implementation

That's (almost) all !
Thanks for participating!

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

FastFlow project site:



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

EU-FP7 projects using FastFlow: