GridKa School 2014

Big Data, Cloud Computing and Modern Programming

Parallel Programming using FastFlow

Massimo Torquati <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy

Karlsruhe, September 2nd, 2014

Outline

- Structured Parallel Programming
  - Algorithmic Skeletons & Parallel Design Patterns
- FastFlow: A data-flow framework for heterogeneous many-core platforms
  - In this talk we consider mainly multi-core systems
- Applications developed using FastFlow & Structured Parallel Programming

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

Structured Parallel Programming

- The parallel programmer is relieved from all concerns related to the implementation of the parallel paradigms on the target platform
- The parallel programmer has to concentrate "*only*" on computational aspects

Separation of concerns principles

Skeletons & Patterns

Algorithmic Skeletons	Parallel Design Patterns

Algorithmic Skeletons

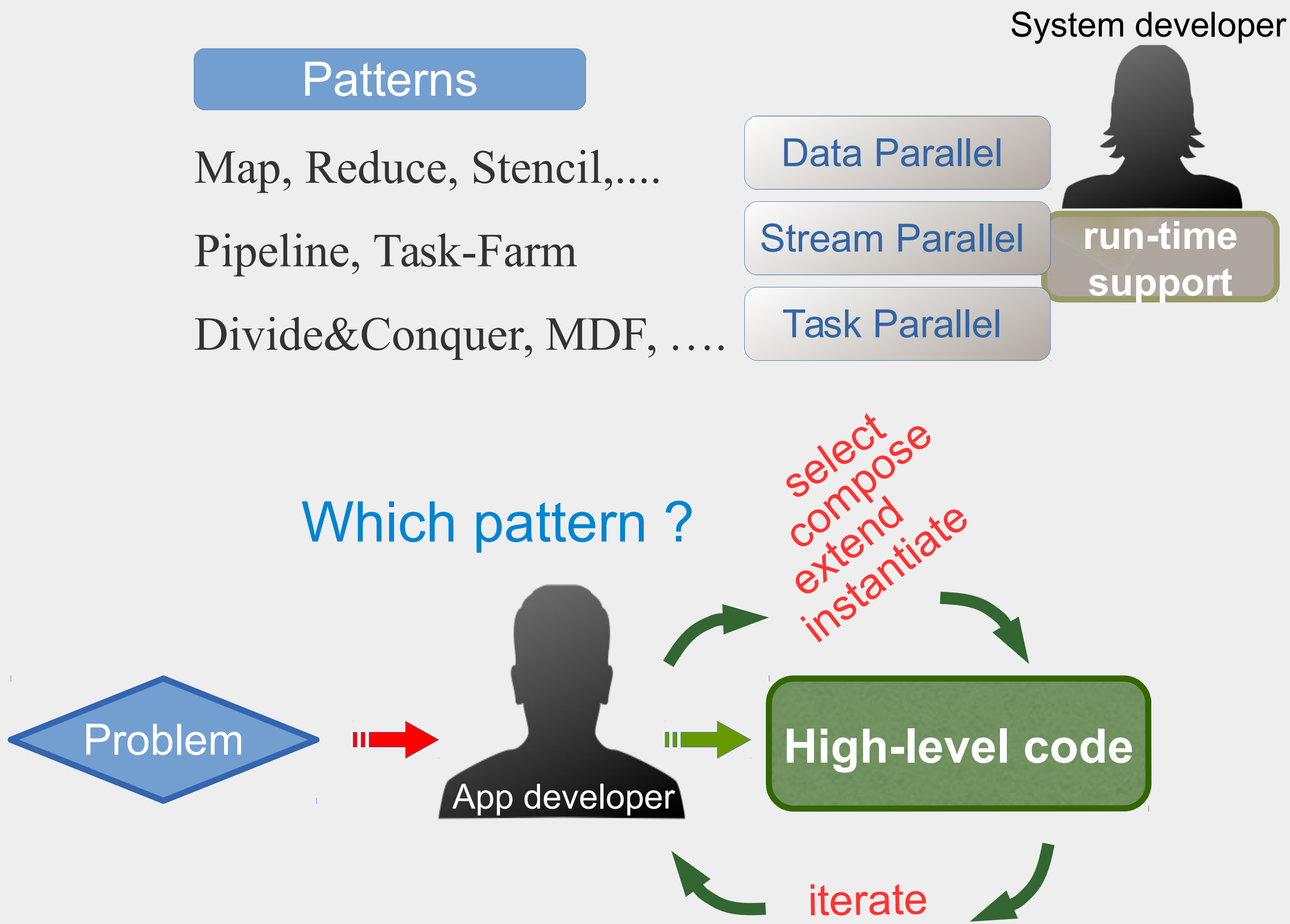
- From HPC community
- From early '90
- Pre-defined parallel high-order functions provided as constructs or lib calls

Parallel Design Patterns

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

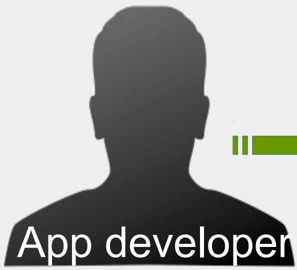
- *The same concept at different abstraction levels*
- We use the two terms patterns and skeletons, interchangeably.
  - We want to emphasise the similarities of these two concepts

Structured Parallel Programming using Patterns



Structured Parallel Programming: example

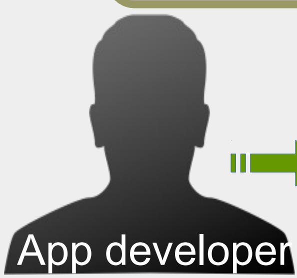
- Problem: apply the function F to all N elements of the input array A



- Select the Map pattern
- Instantiate the Map pattern with ***static partitioning***

run-time support

- Select the "best" Map implementation on the target platform with *static partitioning* of data
- Execute the code and provide the result



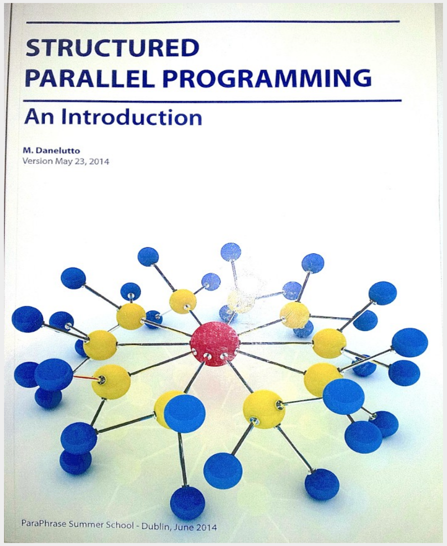
- Analyse the result: the workload is unbalanced
- Instantiate the Map pattern with ***dynamic partitioning***

run-time support

- Select the "best" Map implementation on the target platform with *dynamic partitioning* of data
- Execute the code and provide the result

Assessment (algorithmic skeletons)

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.

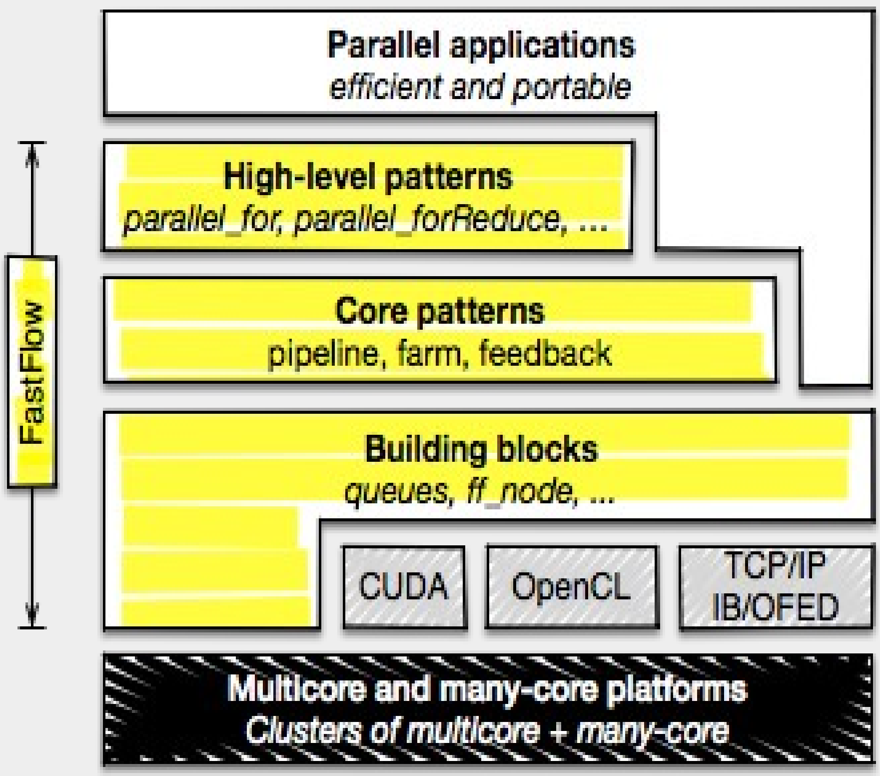


"Structured Parallel Programming" by Marco Danelutto

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

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

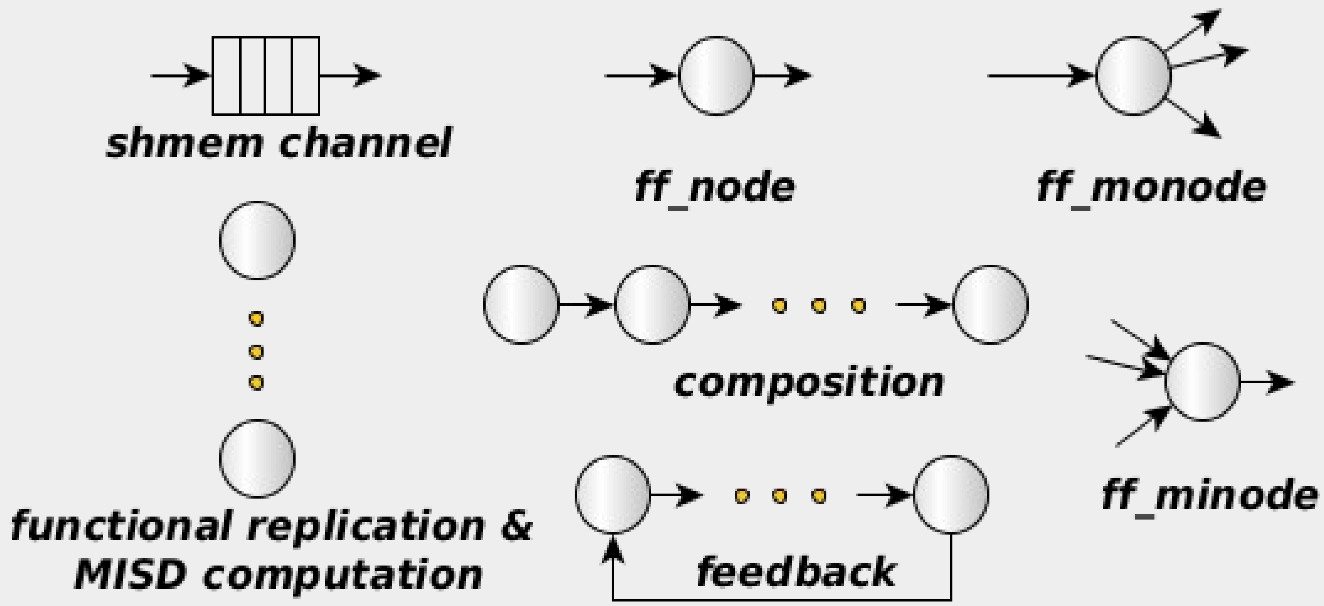
The FastFlow framework



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

- C++ class library
- **Promotes structured parallel programming**
- It aims to be flexible and efficient enough to target ***multi-core***, **many-core** and ***distributed*** 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

FastFlow Building Blocks



Minimal definition of a node :

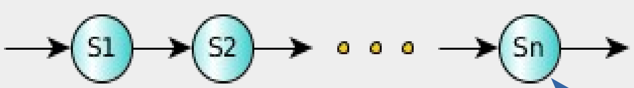
```
struct myNode: ff_node {
  void *svc(void *task) { … return task; }
};
```

- nodes are concurrent activities
  - POSIX threads
- arrows are shared-memory channels
  - implemented as *SPSC lock-free queues*

Stream Parallel Patterns (“core” patterns)

A stream is a sequence of data items having the same type

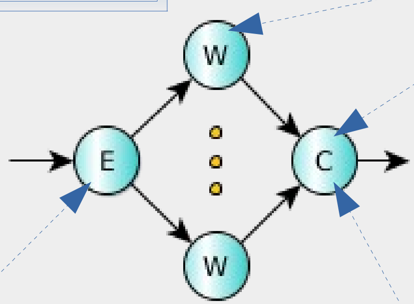
pipeline




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

ff_node

task-farm



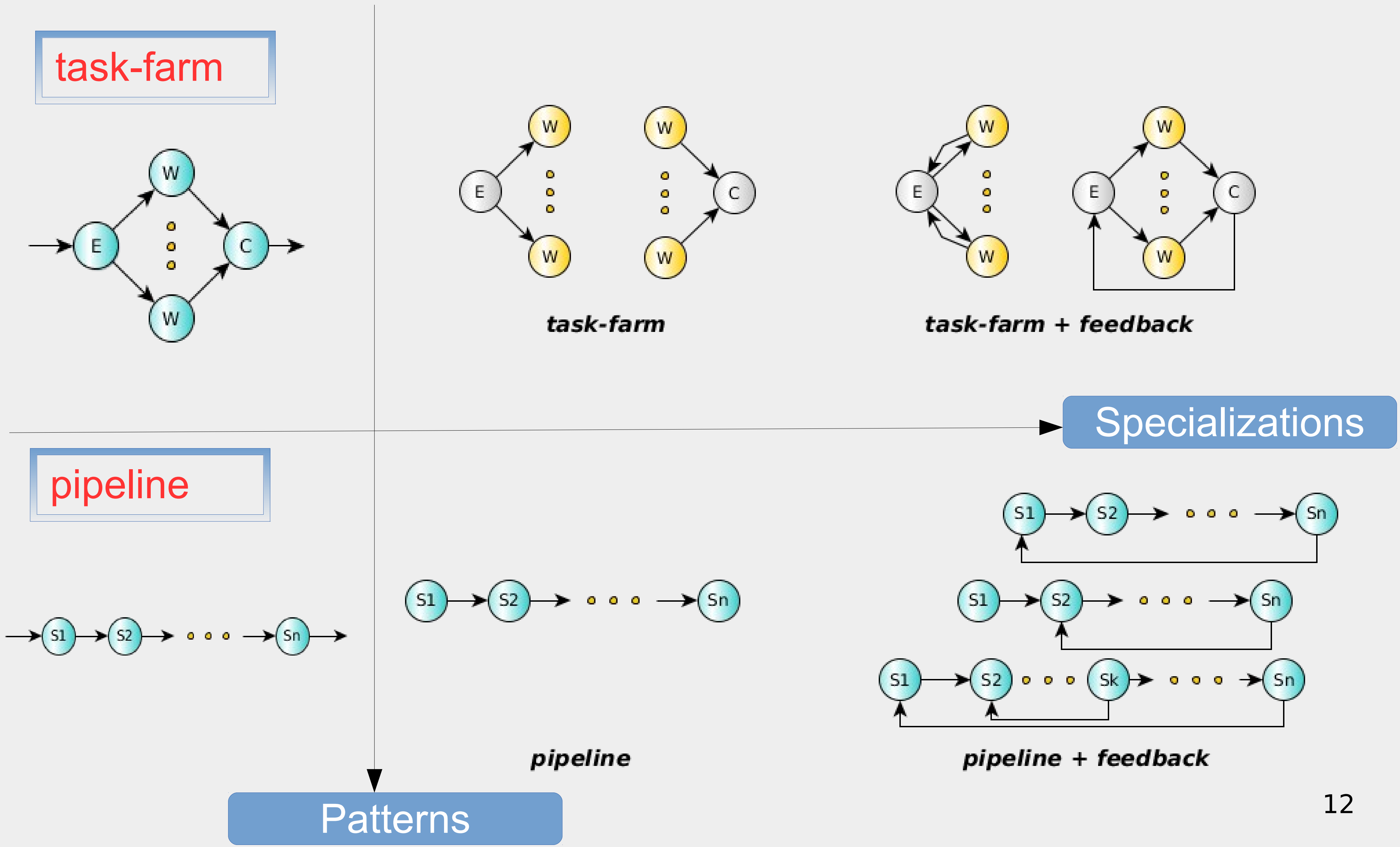

```
std::vector<ff_node*> WorkerArray;

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

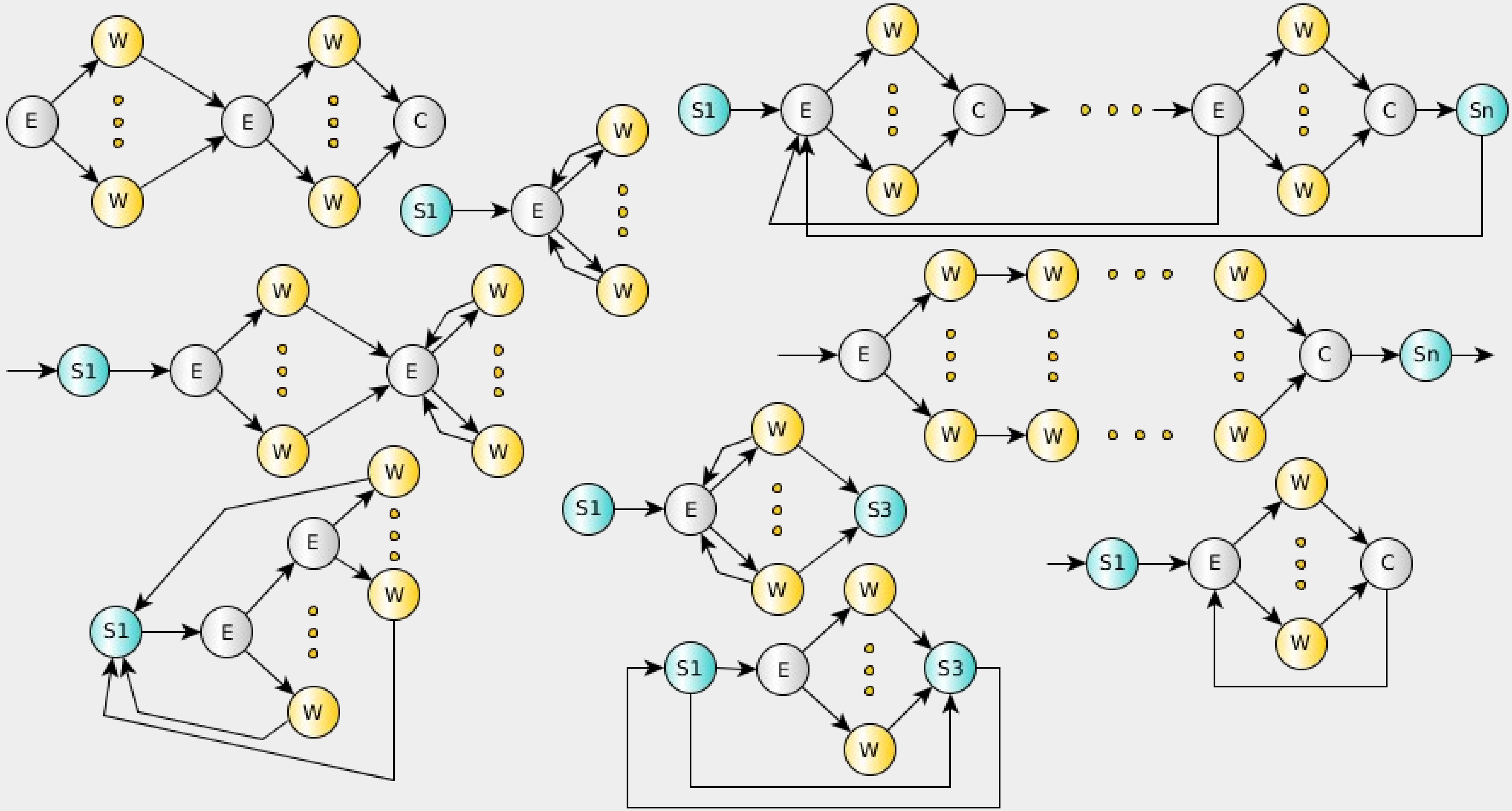
Emitter:
schedules input data items

Collector:
gathers results

FastFlow “core” patterns

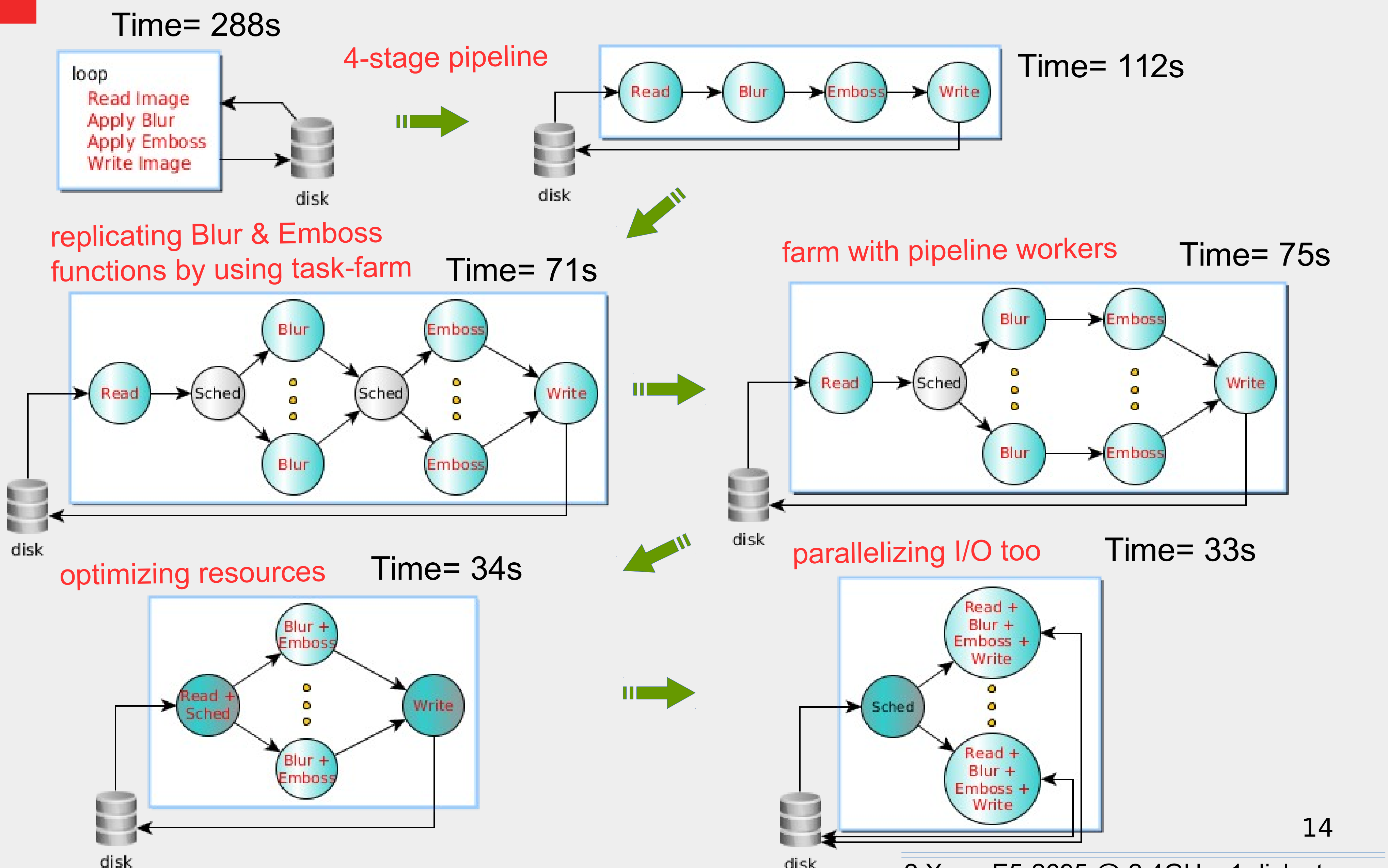


Core Patterns Composition



pipeline + task-farm + feedback

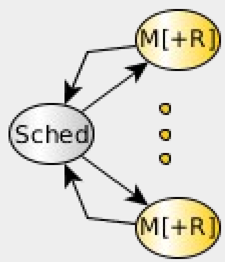
Example: filtering images



2 Xeon E5-2695 @ 2.4GHz, 1 disk storage

Data Parallel Patterns (1)

ParallelFor/ParallelForReduce



M: map function

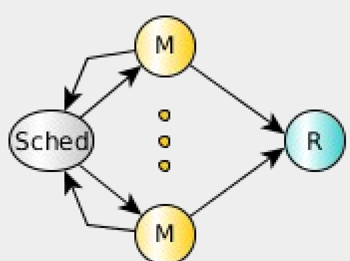
R: reduce function

M & R: computed in parallel

```
ParallelFor pf;
pf.parallel_for(0,N,[&](const long i) {
   A[i] = F(i);
});
```

```
ParallelForReduce<double> pfr;
pfr.parallel_reduce(sum, 0.0,
                              0,N,[&](const long i) {
                sum += F(i);
}, [](double &sum, const double v) {
                sum += v;
});
```

ParallelForPipeReduce



M & R: computed in pipeline

```
ParallelForPipeReduce<task_t> pfr;
auto MapF = [&](.......ff_buffernode &n) {
 … ;
 n.put(task);
};
auto ReducdF = [&](task_t *task) { … };
pfr.parallel_reduce_idx(0,N,step,chunk,
                                    MapF, ReduceF);
```

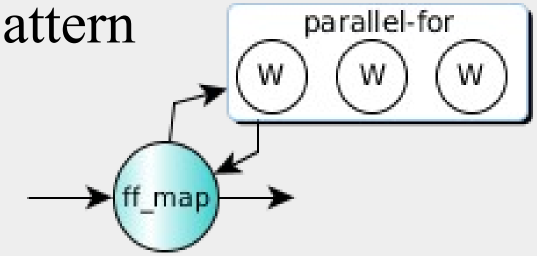
- Static and dynamic scheduling of tasks
- With or without scheduler thread (Sched)

Data Parallel Patterns (2)

Map | Stencil | poolEvolution

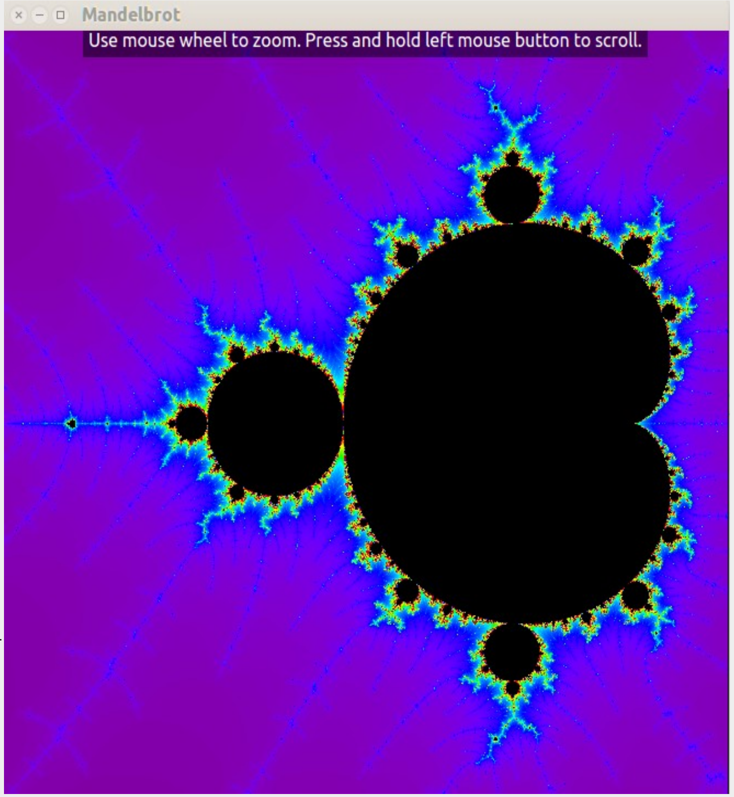
- May be used inside stream parallel patterns (*pipeline* and *task-farm*)
- On multi-core systems all of them are implemented on top of ParallelFor* patterns
  - Algorithms + ParallelFor* wrappers
- For example, the Map pattern is a FastFlow node with inside an “optimized instantiation“ of the ParallelForReduce pattern

```
struct myMap: ff_map<> {
  void *svc(void *task) { ….;
       ff_map<>::parallel_for(.....);
}};
```



- Targeting GPGPUs: support for both CUDA and/or OpenCL available for some data-parallel patterns
  - Work still in progress

Example: Mandelbrot set (1)



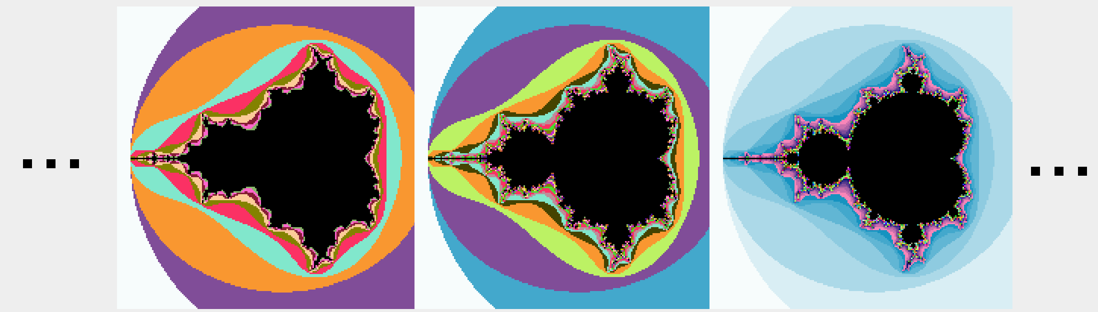
- Very simple data-parallel computation
  - Each pixel can be computed independently
  - Simple ParallelFor implementation
- Black-pixel requires much more computation
- A naïve partitioning of the images quickly leads to load unbalanced computation and poor performance
  - Let's consider the minimum computation unit a single image line (image size 2048x2048, max 10^3 iterations per point)
    - Static partitioning of lines (48 workers) **MaxSpedup 14**
    - Dynamic partitioning of lines (48 workers) **MaxSpeedup 37**

Data-partitioning may have a big impact on the performance

2 Xeon E5-2695 @ 2.4GHz, 1 disk storage

Example: Mandelbrot set (2)

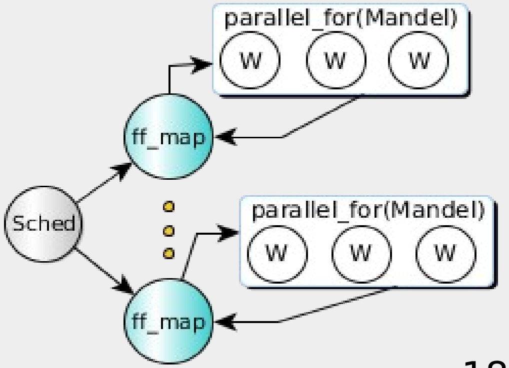
- Suppose now we want to compute a number of Mandelbrot images (for example varying the computing threshold per point)



- 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));
```

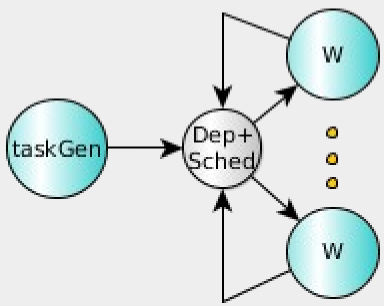
- Which one is better having limited resources ?
  - Depends on many factors, *too difficult to say in advance*



Moving quickly between the two solutions is the key point

Task-parallel patterns

Macro-Data Flow (MDF)

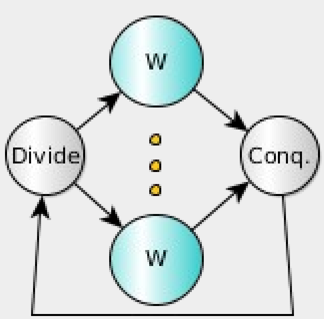


pipeline(taskGen, farm(Ws,Sched))

- The MDF executes data-dependency graph (DAG)
- Is a general approach to parallelism
- The user has to specify data dependency using a sequential function "taskGen" and to generate tasks (by using the *AddTask* method)
- The run-time automatically takes care of dependencies and then schedules ready task to Ws

```
void taskGen( mdf ....) {
   Param 1 = {&A, INPUT};
   Param 2 = {&A, OUTPUT};
   mdf->AddTask(Params, F, A);
}
ff_mdf<Params> mdf(taskGen);
mdf.run_and_wait_end();
```

Divide&Conquer



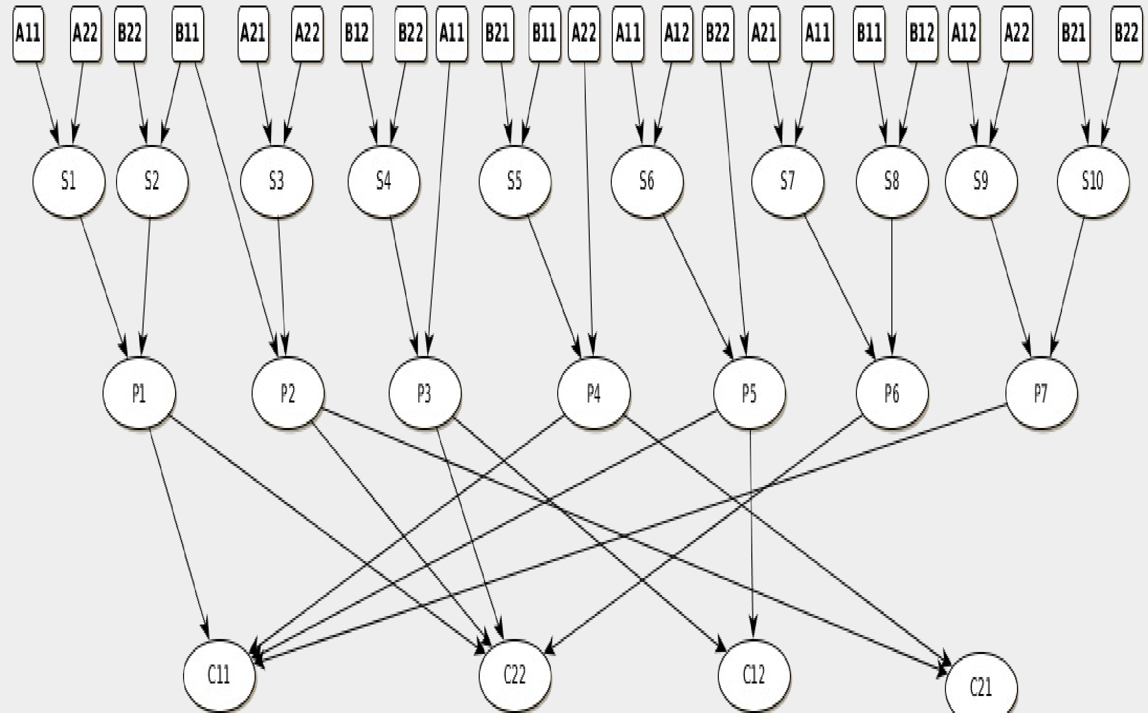
- Currently is a task-farm with feedback channel with specialized Emitter and Collector.

Example: Strassen's algorithm

- Matrix multiplication using Strassen's algorithm:

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} X \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

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 function *taskGen* is responsible for generating instructions S1, S2, P1, S3, P2 ….... in any order specifying INPUT and OUTPUT dependencies.
- Each macro instruction can be computed in parallel using a ParallelFor pattern or optimized linear algebra matrix operations (BLAS, PLASMA, Lapack...)

Real applications (some)

Stream Parallel

- Bowtie (BT) and BWA Sequence Alignment Tools
- Peafowl, an open-source parallel DPI framework

Task Parallel

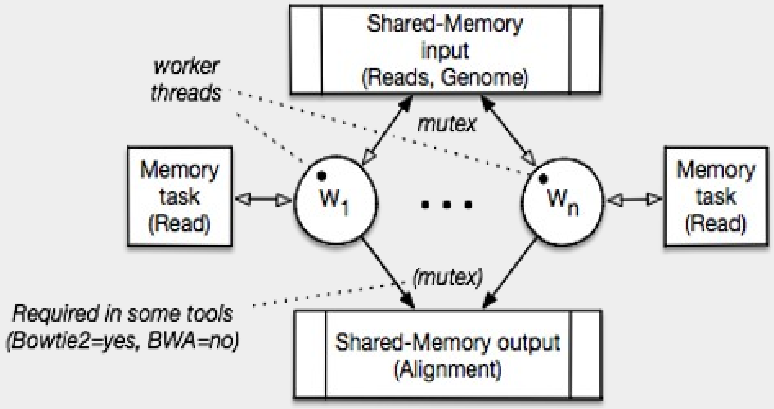
- YaDT-FF: fast C4.5 classifier (*)
- Block-based LU & Cholesky factorizations

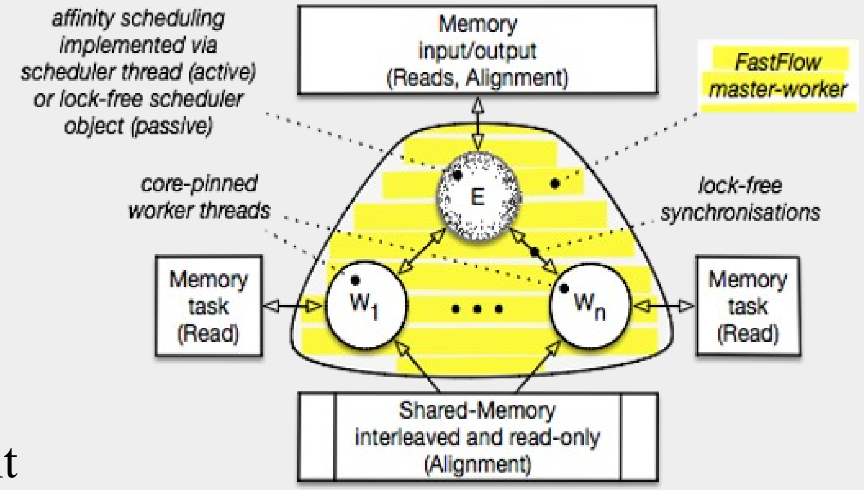
Data Parallel

- Two Stage Image and Video Denoiser

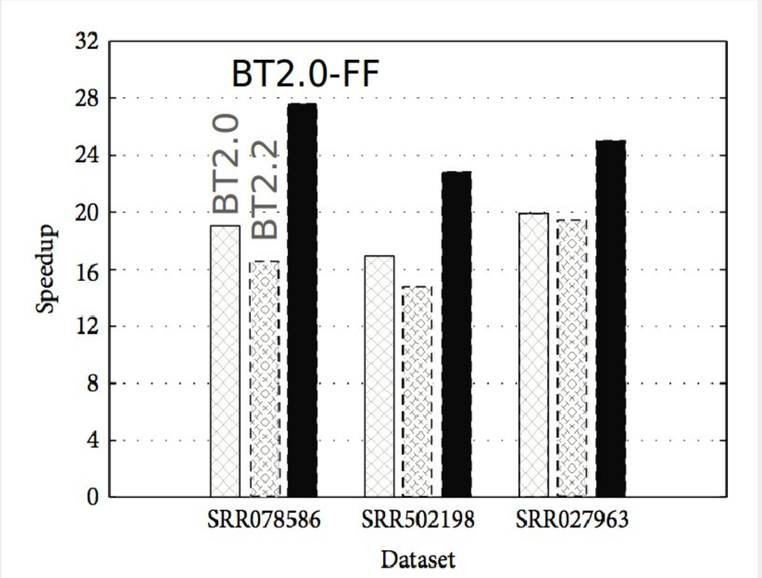
(*) M. Aldinucci, S. Ruggieri and M. Torquati "Decision Tree building on muli-core using FastFlow", Concurrency and Computations: practice and experience, Vol. 26, 2014

Stream: Bowtie (BT) and DWA Sequence Alignment Tools





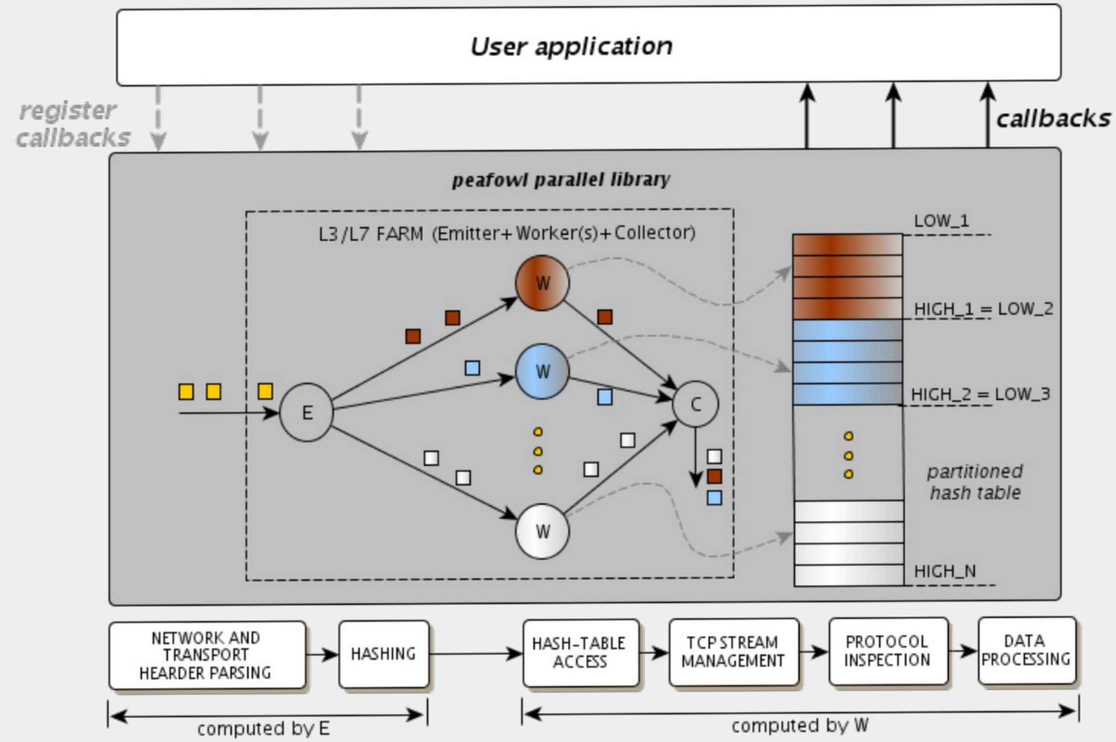
- Very widely used tools for DNA alignment
- Hand-tuned C/C++/SSE2 code
- Spin-locks + POSIX Threads
- Reads are streamed from memory-mapped files to worker threads



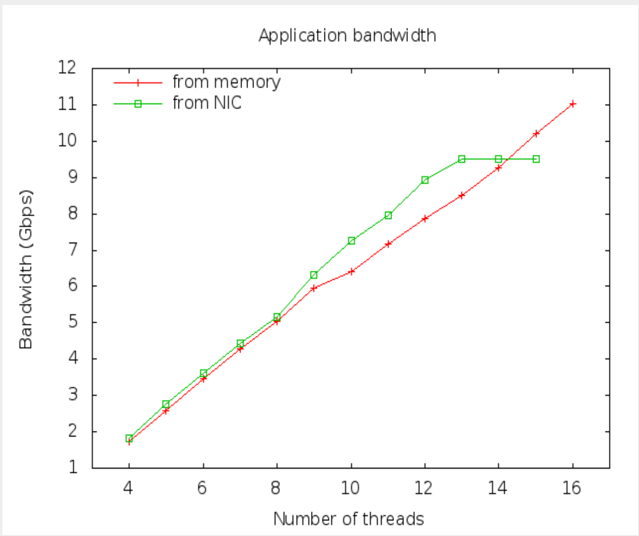
- Task-farm+feedback implementation in FastFlow
- Thread pinning + memory affinity + affinity scheduling
- Quite substantial improvement in performance

C. Misale, G. Ferrero, M. Torquati, M. Aldinucci "Sequence alignment tools: one parallel pattern to rule them all?" BioMed Research International, 2014

Stream: 10Gbit Deep Packet Inspection (DPI) on multi-core using Peafowl



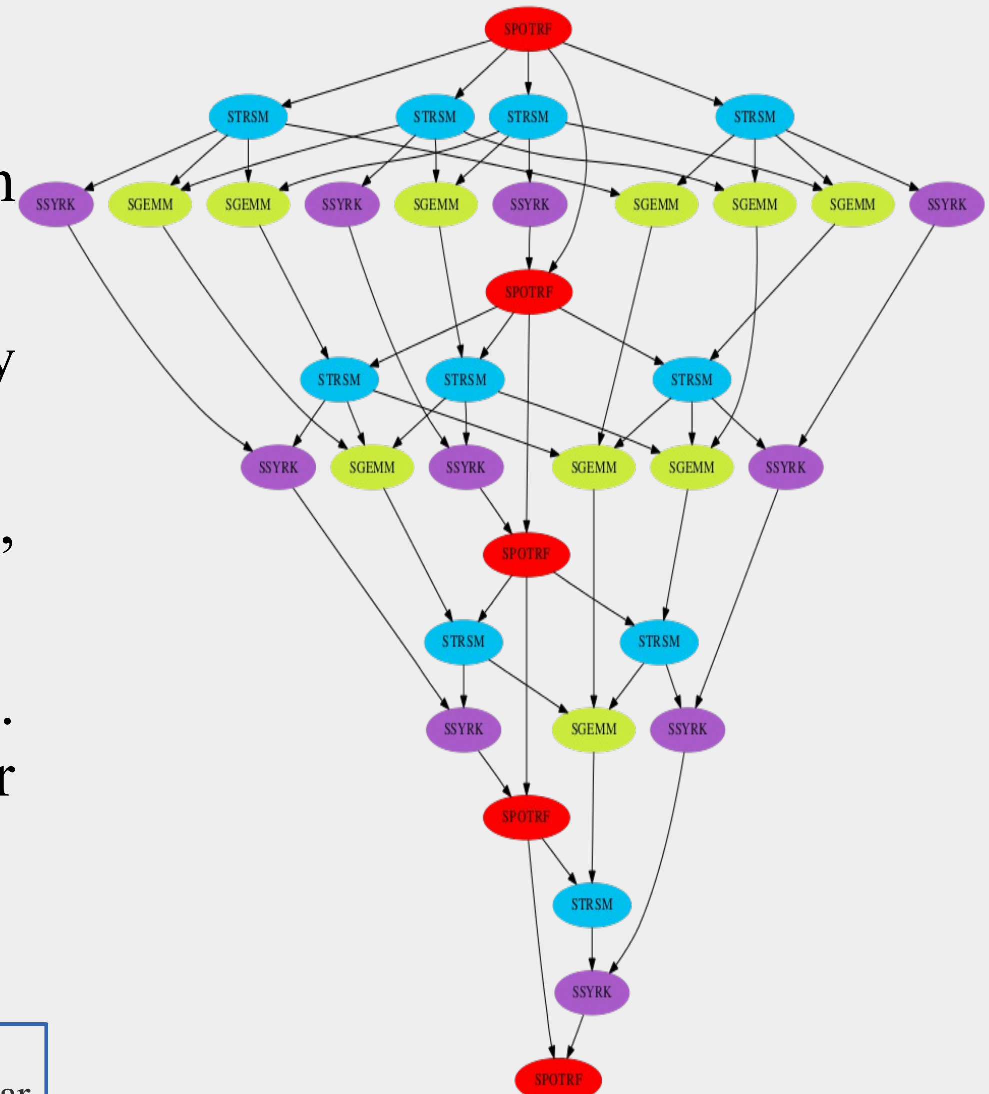
- Peafowl is an open-source high-performance DPI framework with FastFlow-based run-time
  - Task-farm + customized Emitter and Collector
- We developed an HTTP virus pattern matching application for 10 Gibit networks
- It is able to sustain the full network bandwidth using commodity HW



M. Danelutto, L. Deri, D. De Sensi and M. Torquati “Deep Packet Inspection on commodity hardware using FastFlow” in PARCO 2013 conference, Vol. 25, pg. 99-99, 2013

Task-Parallel: LU & Cholesky factorizations using the MDF pattern

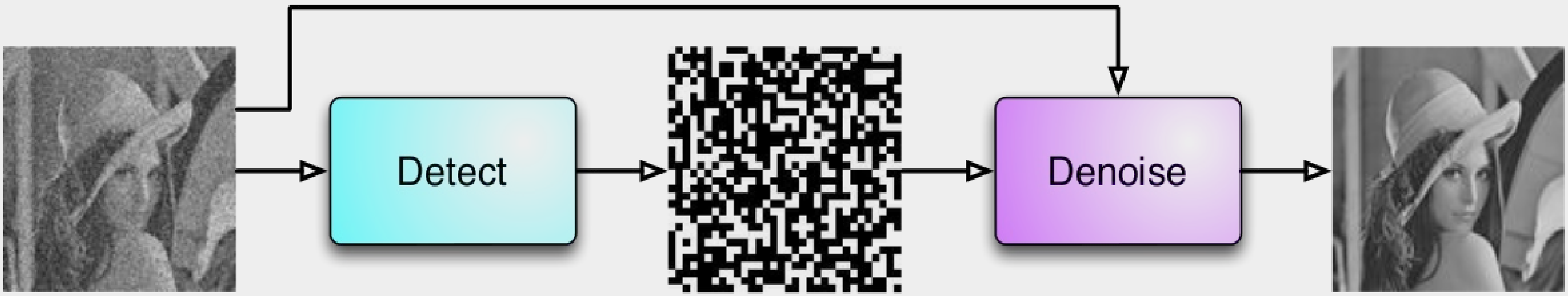
- Dense matrix, block-based algorithms
- Macro-Data-Flow (MDF) pattern encoding dependency graph (DAG)
  - The DAG is generated dynamically during computation
- Configurable scheduling of tasks, affinity scheduling
- Comparable performance w.r.t. specialized multi-core dense linear algebra framework (PLASMA)



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

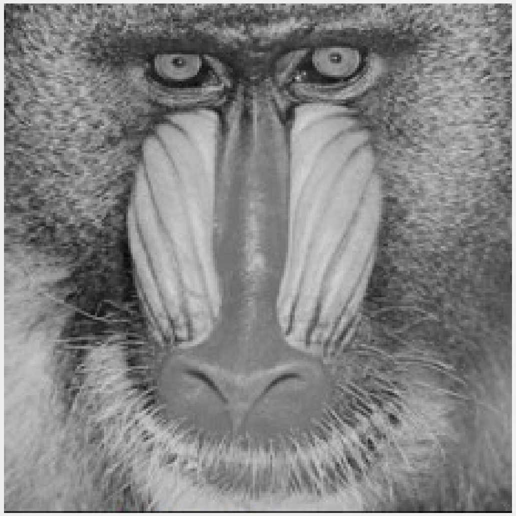
Data-Parallel: Two stage image restoration



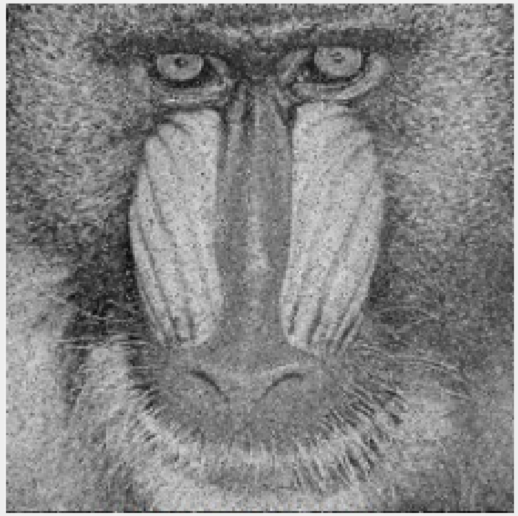
- Detect: adaptive median filter, produces a noise map
- Denoise: variational Restoration (iterative optimization algorithm)
  - 9-point stencil computation
- High-quality edge preserving filtering
- Higher computational costs w.r.t. other edge preserving filters
  - without parallelization, no practical use of this technique because too costly
- *The 2 phases can be pipelined for video streaming*

M. Aldinucci, C. Spampinato, M. Drocco, M. Torquati and S. Palazzo "A parallel edge preserving algorithm for salt and pepper image denoising" IPTA 2012 conference, 2012

Salt & Pepper image restoration

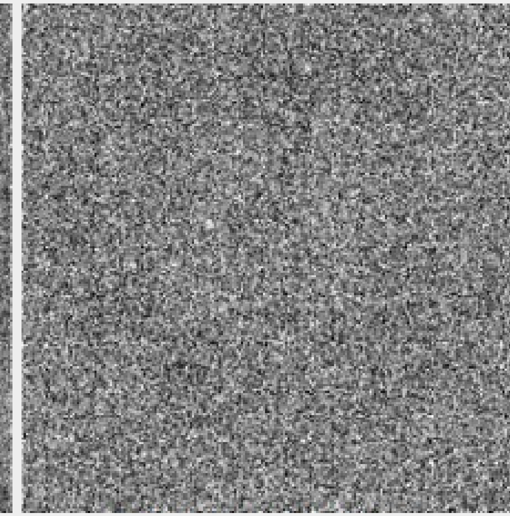


Original Baboon standard test image 1024x1024



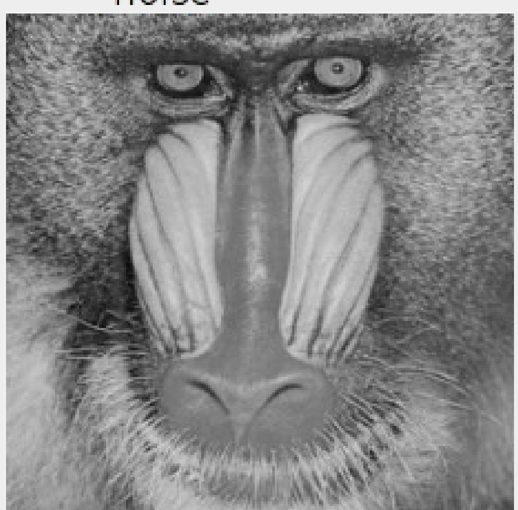
10% impulsive noise

50% impulsive noise

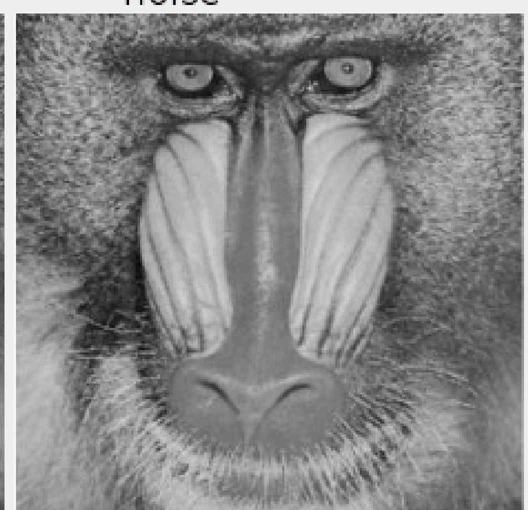


90% impulsive noise

Restored



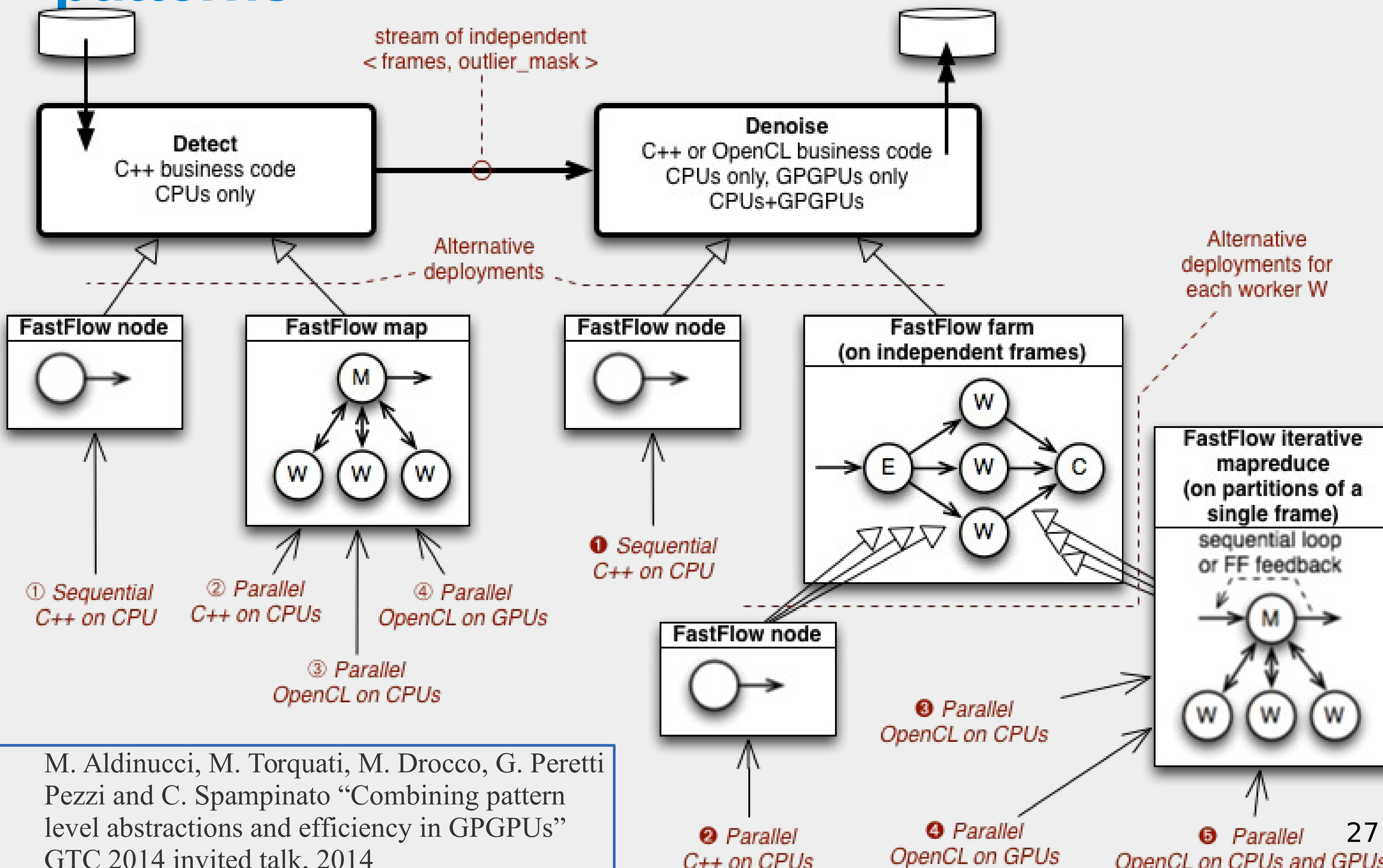
PNSR 43.29dB MAE 0.35



PNSR 32.75dB MAE 2.67

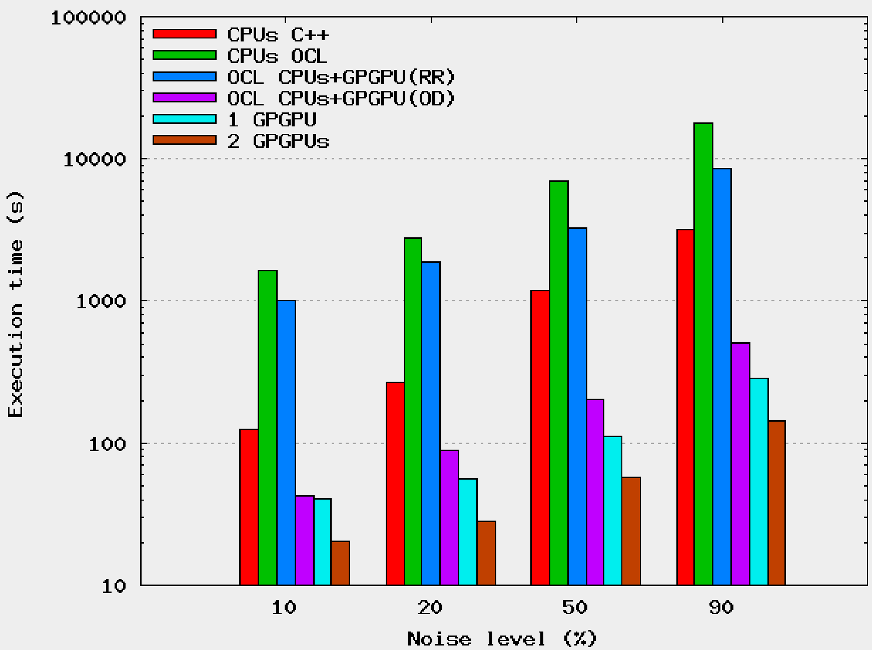
PNSR 23.4 MAE 11.21

Stream+Data-Parallel: Video de-noising, possible parallelization options using patterns



M. Aldinucci, M. Torquati, M. Drocco, G. Peretti Pezzi and C. Spampinato “Combining pattern level abstractions and efficiency in GPGPUs” GTC 2014 invited talk, 2014

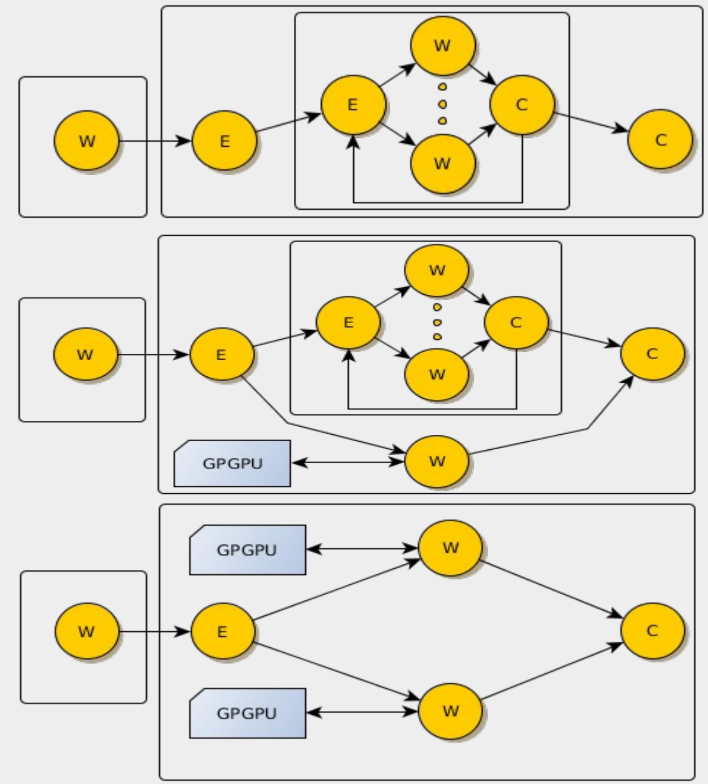
Video de-noising: different deployments



CPU C++ only

OpenCL CPU + GPGPU

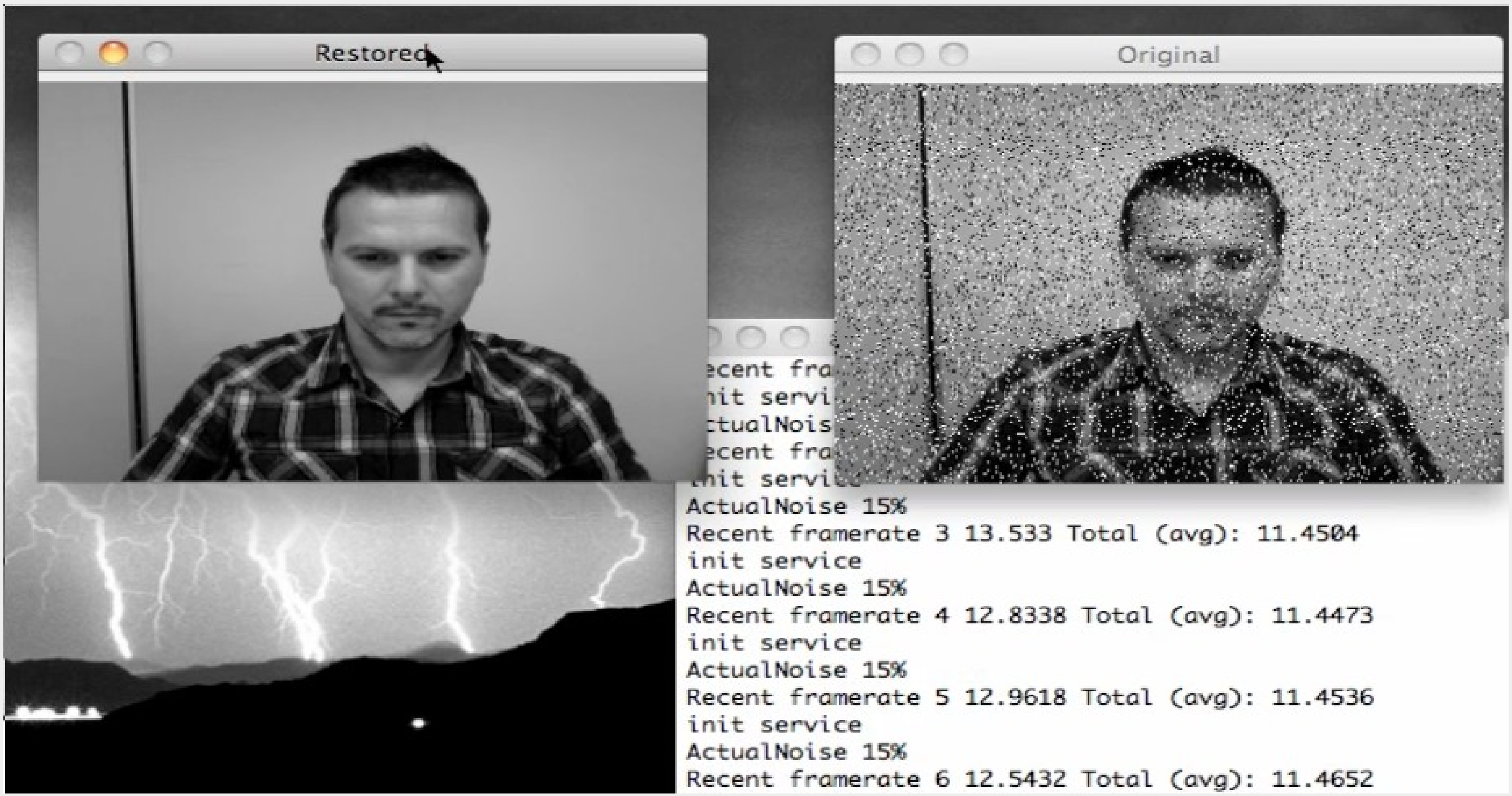
GPGPUs only



- Best option is to use all available GPGPUs

Experiment name	Deployment	Brief description
CPUs C++	①+❷	Detect C++/1CPU, Denoise C++/14CPUs
CPUs OCL	①+❸	Detect C++/1CPU, Denoise OpenCL/14CPUs
OCL CPUs+GPGPU(RR)	①+❺	Detect C++/1CPU, Denoise OpenCL 14CPUs+1GPGPU, Round-Robin scheduling
OCL CPUs+GPGPU(OD)	①+❺	Detect C++/1CPU, Denoise OpenCL 14CPUs+1GPGPU, On-Demand scheduling
1 GPGPU	①+❹	Detect C++/1CPU, Denoise OpenCL/1GPGPU
2 GPGPUs	①+❹	Detect C++/1CPU, Denoise OpenCL/2GPGPUs

Video de-noising demo



Thanks to Marco Aldinucci for the demo video

Conclusions

- Structured Parallel Programming models have been here from many years
  - *It is proved they work*
  - clear semantics, enforce separation of concerns, allow rapid prototyping and portability of code, almost same performance as hand-tuned parallel code.

- FastFlow: a C++ class library framework
  - *Research project* framework
  - Promotes structured parallel programming
  - Can be used at different levels, with each level providing a small number of building-blocks/patterns with clear parallel semantics

Thanks for your attention!

Thanks to:

Marco Danelutto (Pisa)
Marco Aldinucci (Turin)
Peter Kilpatrick (Belfast)

University of Pisa

University of Turin

Queen's University Belfast

FastFlow project site:

http://mc-fastflow.sourceforge.net

http://calvados.di.unipi.it/fastflow

EU-FP7 projects using FastFlow: