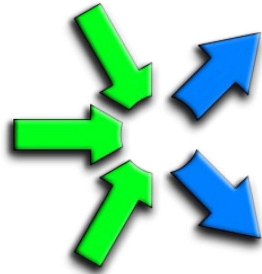


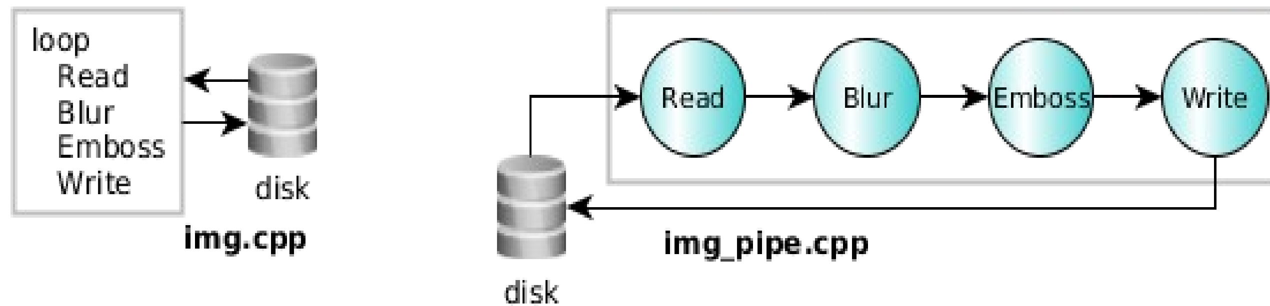
Introduction to FastFlow programming

SPM lecture, December 2015

Massimo Torquati <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy

Parallel pipeline example: image filtering




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

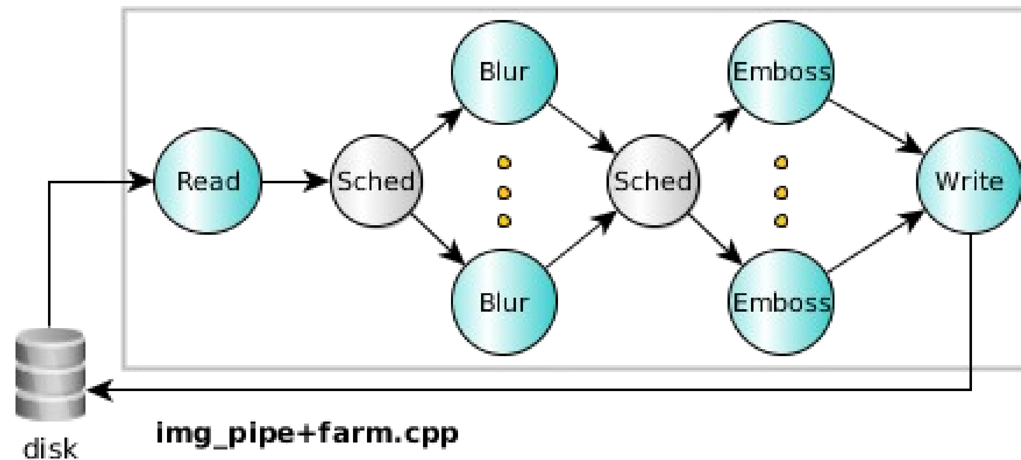
```
// 1st stage
struct Read: ff_node_t<Task> {
  Task *svc(Task *) {
     for(long i=0;i<num_images;++)
        Image *img = new Image;
        Img->read(filename);
        Task *task = new Task(img,filename);
        ff_send_out(task);
  }
  return EOS; // End-Of-Stream
};
```

```
// 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);
}
```

Parallel pipeline example: image filtering



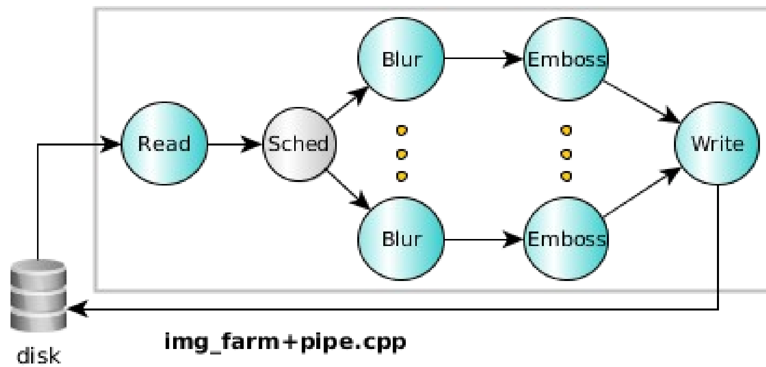

```
ff_Farm<Task> farmBlur(BlurFilter, numBlurWorkers);
farmBlur.remove_collector();
ff_Farm<Task> farmEmboss(EmbossFilter, numEmbosWorkers);
// 4-stage pipeline
ff_Pipe<Task> pipe( read, farmBlur, farmEmboss, write );
pipe.run_and_wait_end();
```

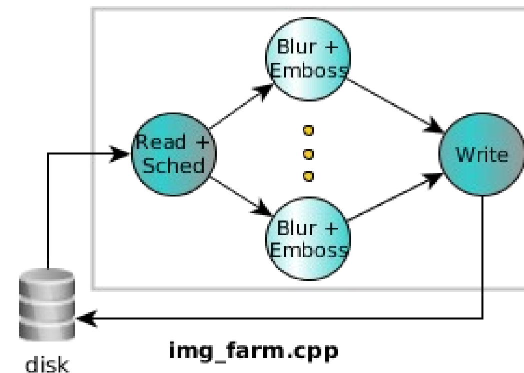
```
// ff_node wrapper to the Write function
struct Writer: ff_minode_t<Task> {
  Task *svc(Task *task) {
     return  Write(task, this);
};
```

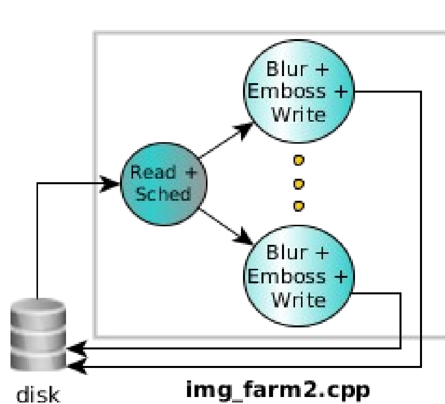
Other nodes are the same as before

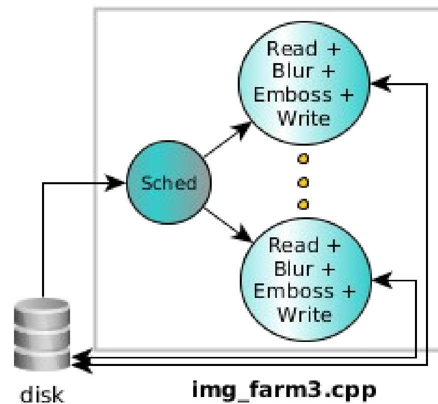
Parallel pipeline example: image filtering

Other simple transformations that require minor modifications









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

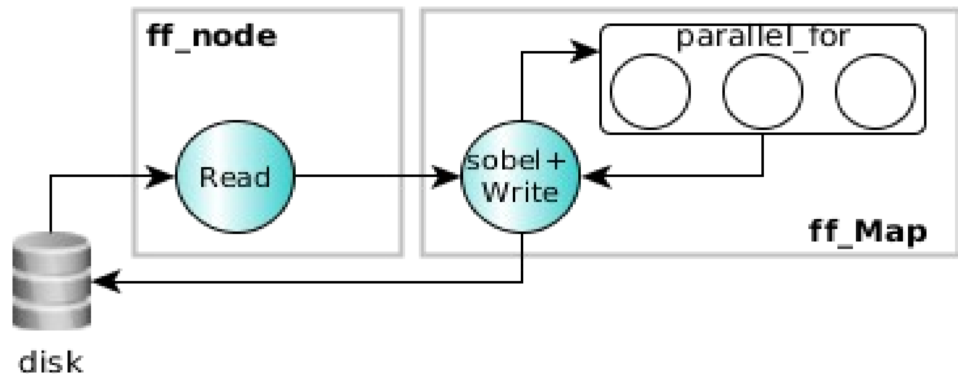
- 320 images of different size (from few kilos to some MB)

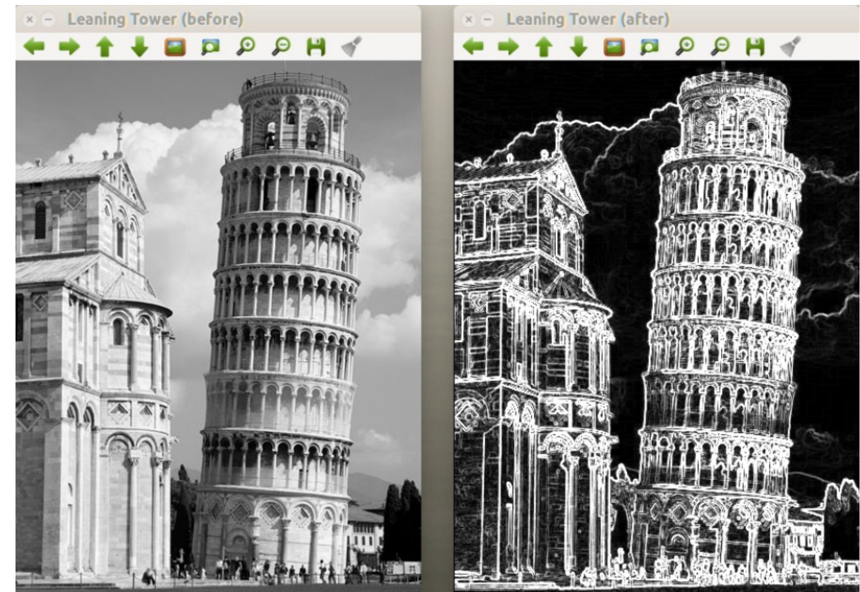
- img (seq): ~ 5m

- img_pipe (4): 2.3m

- img_farm3 (48): 32s

Parallel Pipeline + Data Parallel : Sobel filter



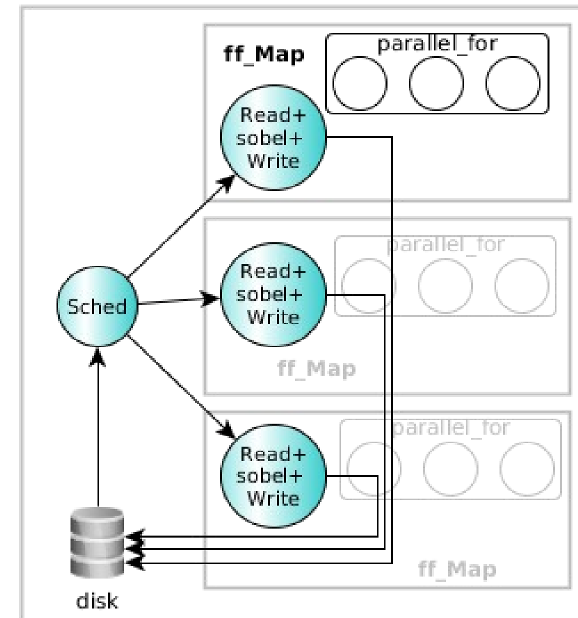



```
struct sobelStage: ff_Map<Task> {
  sobelStage(int mapwks):
     ff_Map<Task>(mapwrks, true) {};

   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

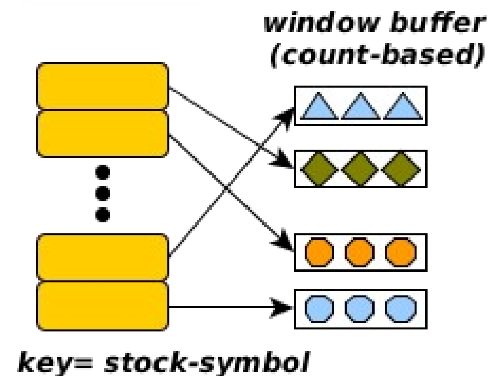
- farm+map (8,4): ~5s
- farm+map (32,1): ~3s

Statefull pipeline

- Consider the following simplified financial application

Sequential pseudo-code:

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



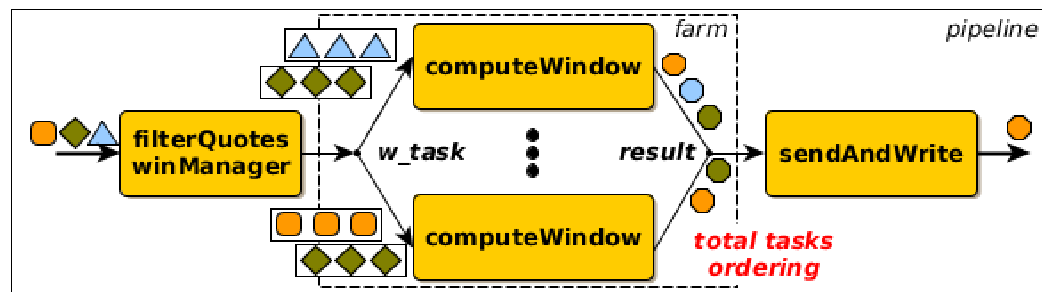
- WindowManagement is based on a hash-table containing different buffers (windows) for each stock symbol

M. Danelutti, T. De Matteis, G. Mencagli and M. Torquati. “Parallelizing High-Frequency Trading Applications by using C++11 Attributes”, REPARA Workshop, ISPA 2015

Statefull pipeline

- The application is logically a 3-stage pipeline (receive, compute, write)

- The middle stage can be replicated (by using an ordered task-farm) if we move the winManagement to the first stage



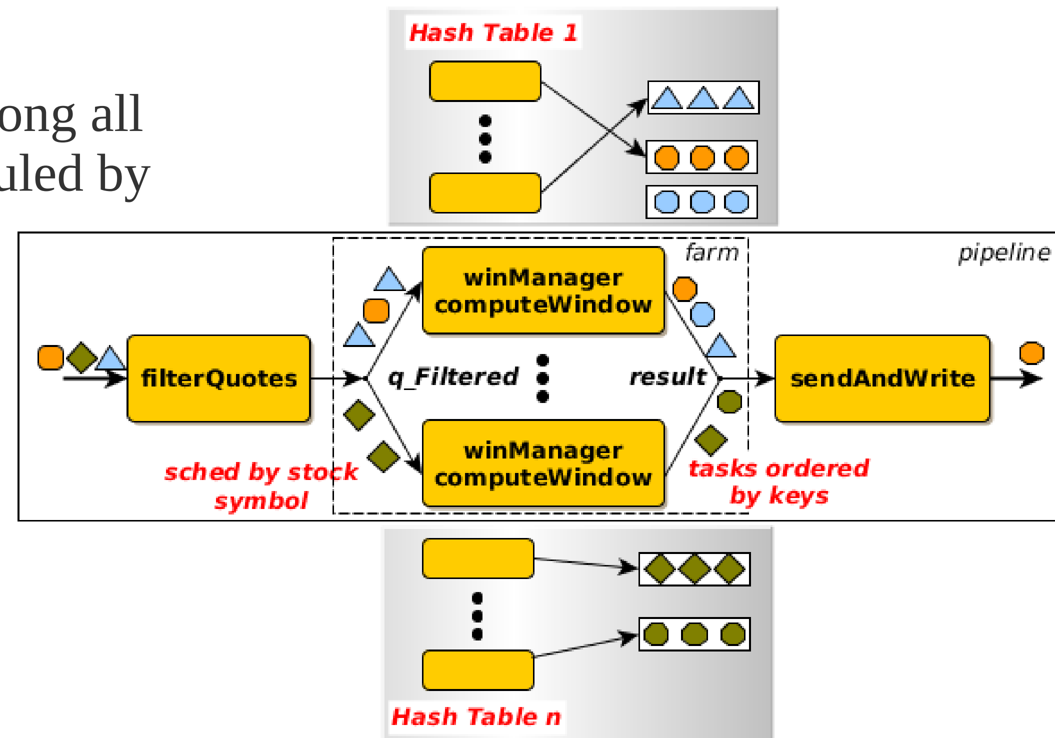
- …. the first stage is (can be) the bottleneck with many workers.
  - WinManager cannot be replicated due to the internal state

Statefull pipeline

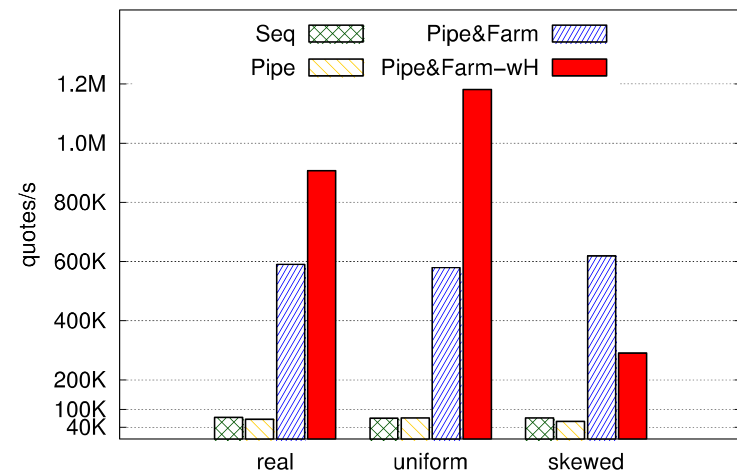
- … but the hast-table can be ***partitioned*** among all workers provided that the quotes are scheduled by stock symbol

Parallel structure:

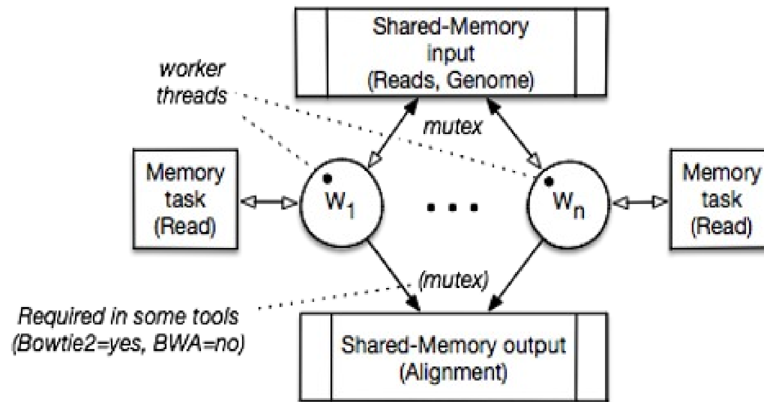
```
struct firstStage: ff_node_t<quote_t> {
  quote_t *svc(quote_t *in) { return filter(*in); }
};
….
Receiver rec(port);
firstStage  first(rec);

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();
```

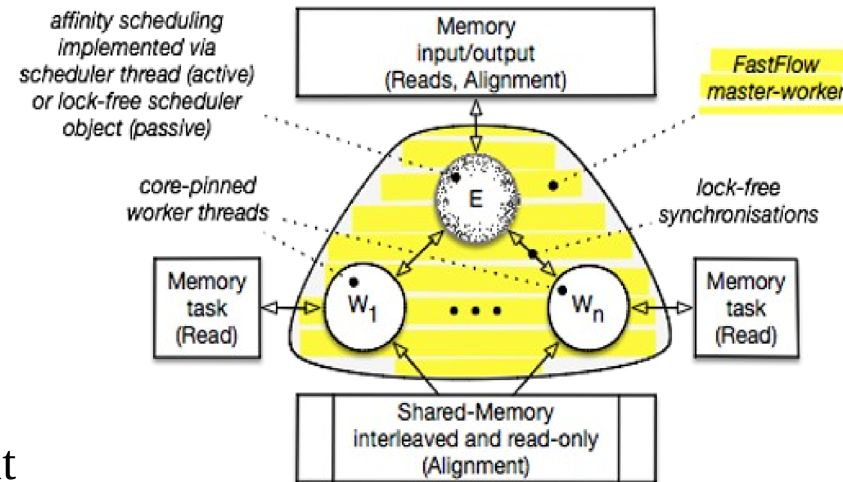


Potential load balancing problems !



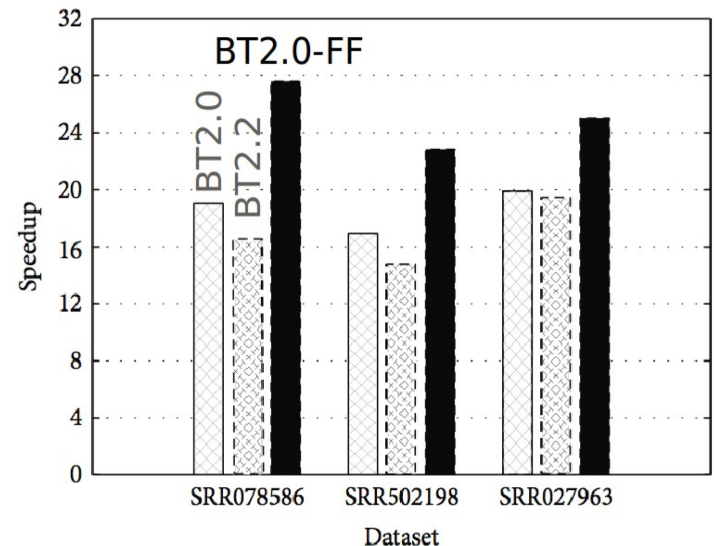
Bowtie (BT) and DWA Sequence Alignment Tools





- 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

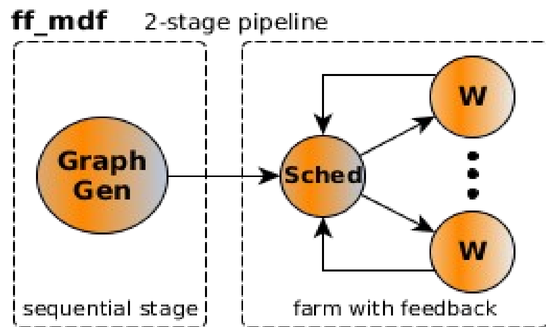
- Task-farm+feedback implementation in FastFlow
- Thread pinning + memory affinity + affinity scheduling
- Quite substantial improvement in performance

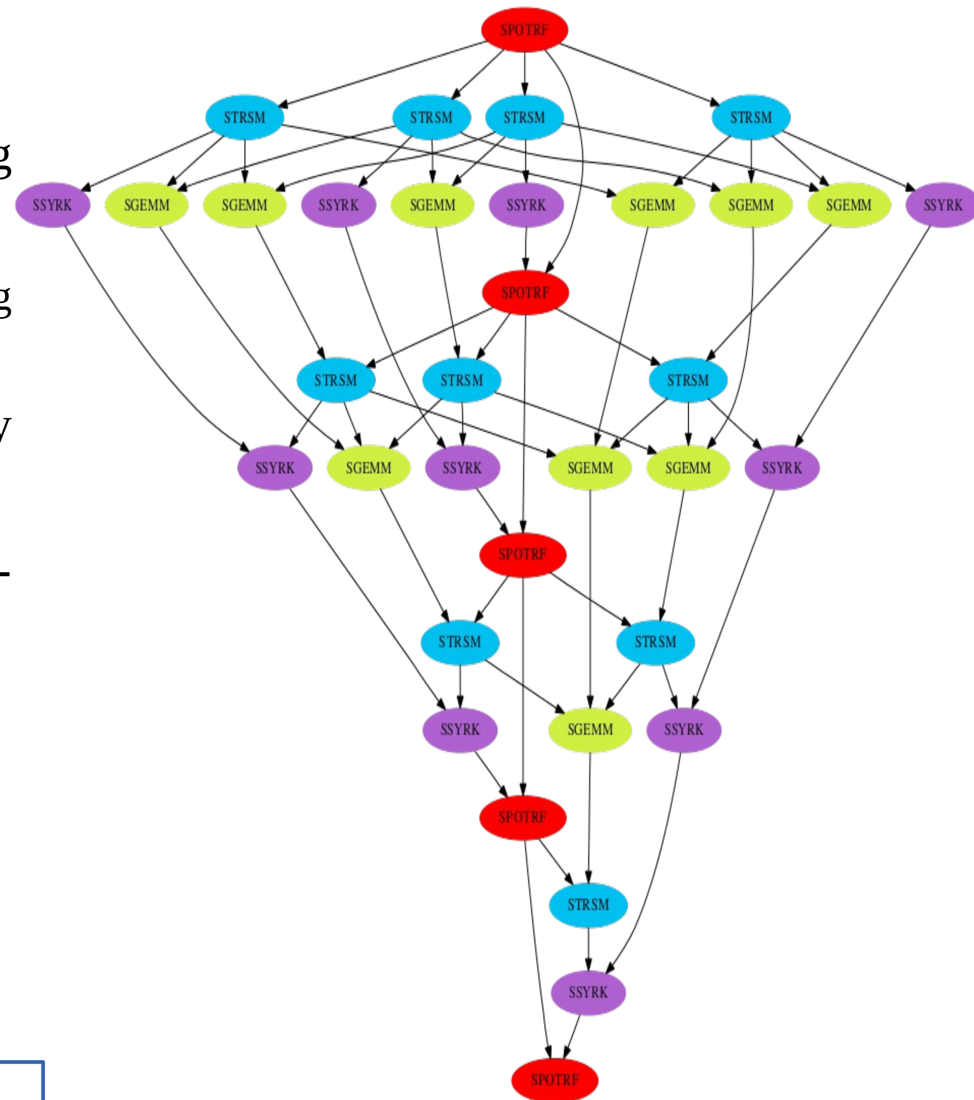


C. Misale, G. Ferrero, M. Torquati, M. Aldinucci “Sequence alignment tools: one parallel pattern to rule them all?“ BioMed Research International, 2014

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)

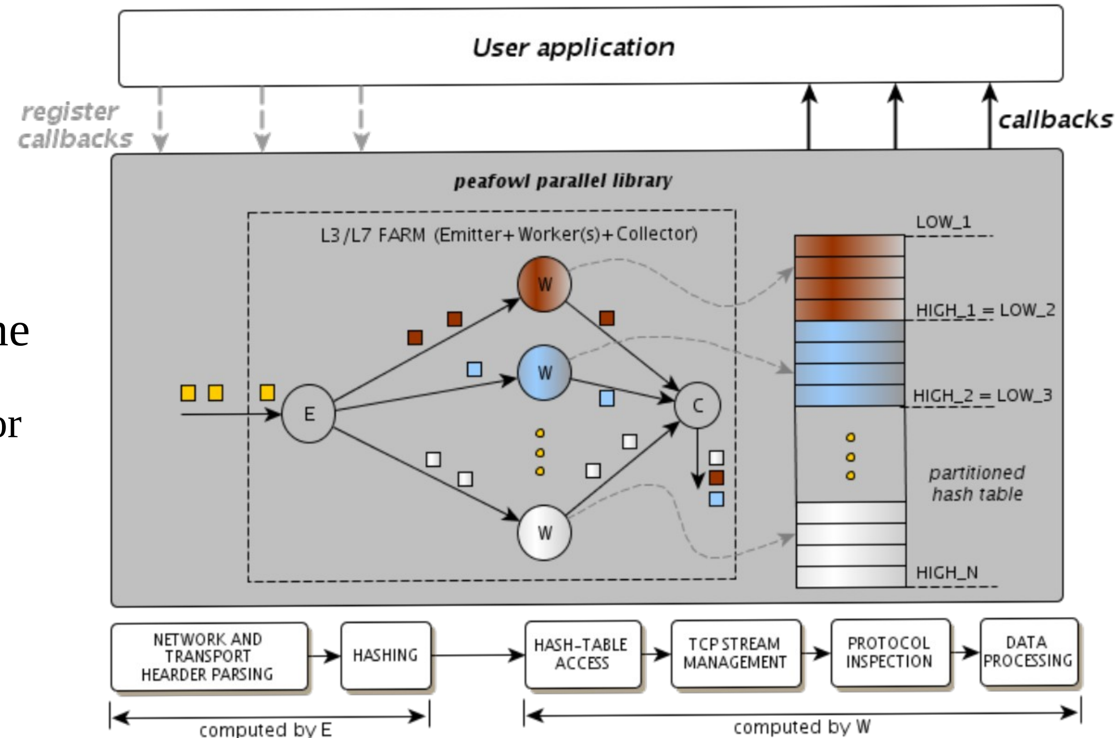




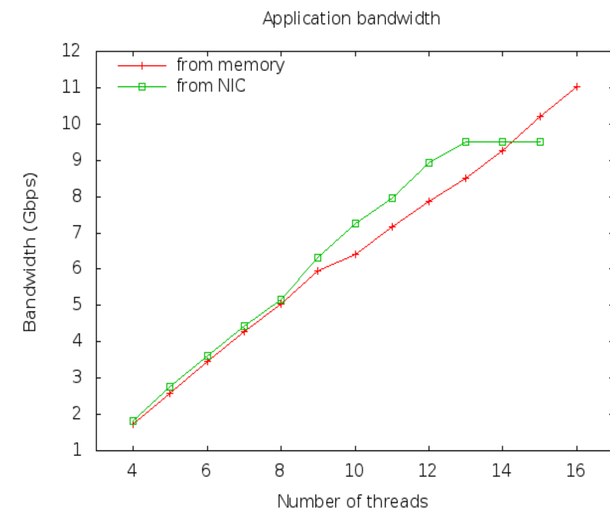
DAG represents, 5 tiles, left-looking version of Cholesky algorithm

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

10Gbit Deep Packet Inspection (DPI) on multi-core

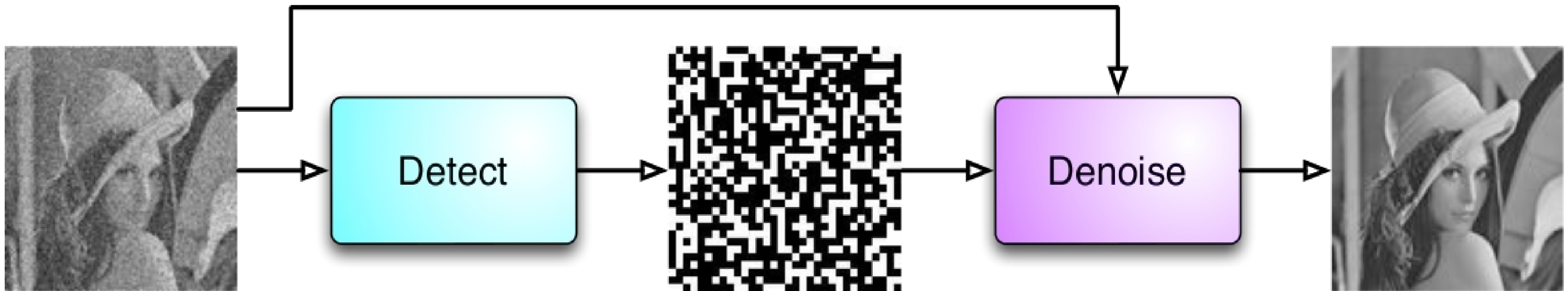


- Peafowl is an open-source high-performance DPI framework with FastFlow-based run-time
  - Task-farm + customized Emitter and Collector
- We developed an HTTP virus pattern matching application for 10 Gibit networks
- It is able to sustain the full network bandwidth using commodity HW



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

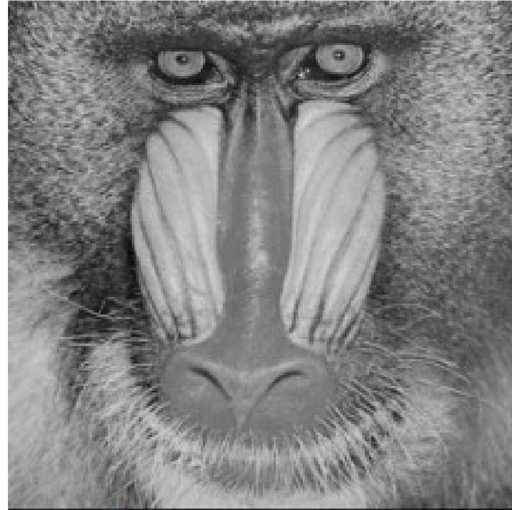
Two stage image restoration



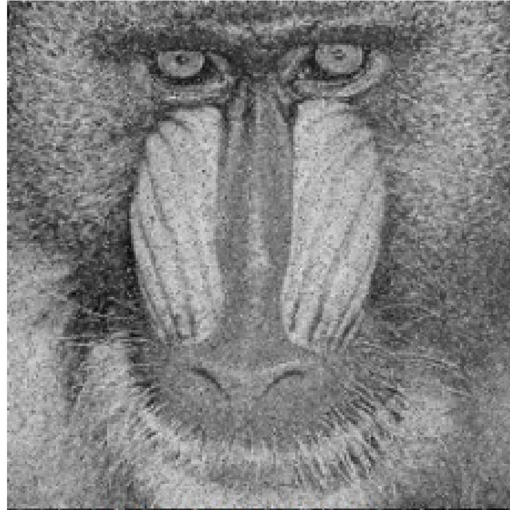
- Detect: adaptive median filter, produces a noise map
- Denoise: variational Restoration (iterative optimization algorithm)
  - 9-point stencil computation
- High-quality edge preserving filtering
- Higher computational costs w.r.t. other edge preserving filters
  - without parallelization, no practical 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

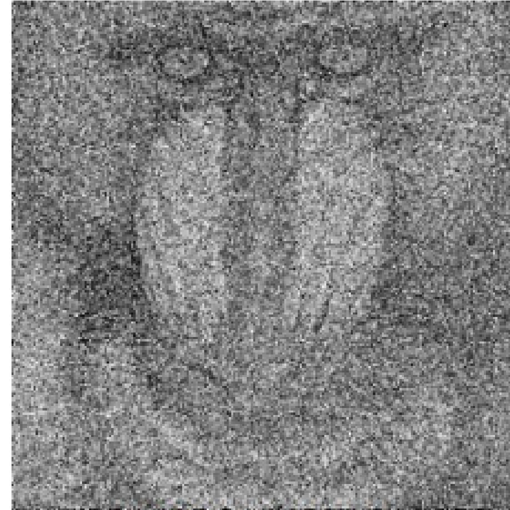
Two stage image restoration: Salt & Pepper image



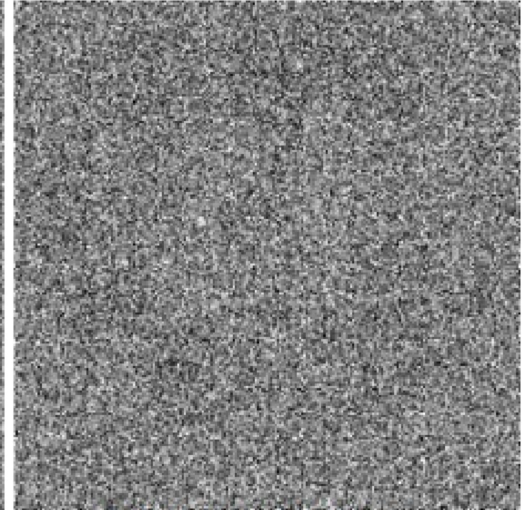
Original Baboon standard test image 1024x1024



10% impulsive noise

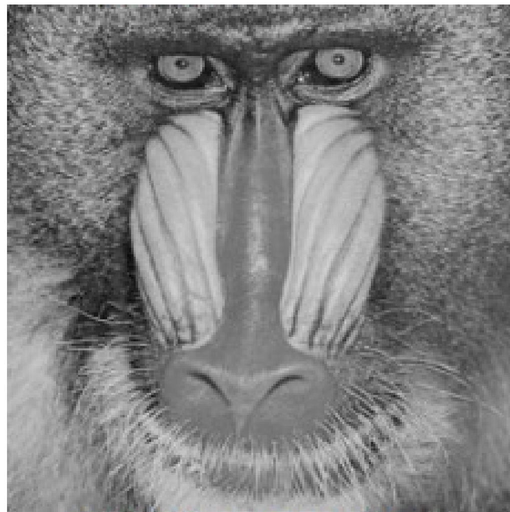


50% impulsive noise

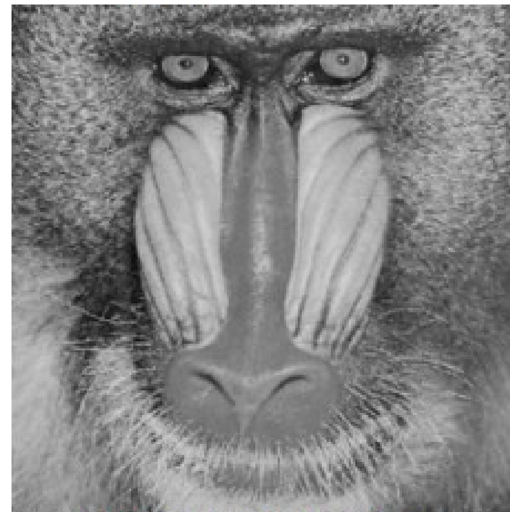


90% impulsive noise

Restored



PNSR 43.29dB MAE 0.35



PNSR 32.75dB MAE 2.67

PNSR 23.4 MAE 11.21