

Introduction to FastFlow programming

SPM lecture, November 2016

Massimo Torquati <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy

Data Parallel Computations

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

FastFlow ParallelFor

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

```
// A and B are 2 arrays of size N  
  
for(long i=0; i<N; ++i)  
    A[i] = A[i] + B[i];
```

```
#include <ff/parallel_for.hpp>  
using namespace ff;  
  
ParallelFor pf; // defining the object  
  
pf.parallel_for(0, N, 1, [&A,B](const long i) {  
    A[i] = A[i] + B[i];  
});
```

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

FastFlow ParallelForReduce

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

- Example:

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

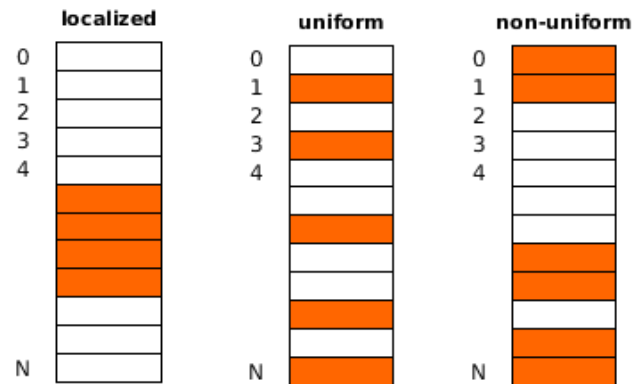
```
#include <ff/parallel_for.hpp>
using namespace ff;

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

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

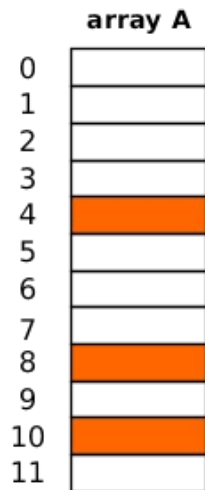
Iterations scheduling

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



- Problem: how to schedule loop iterations ?

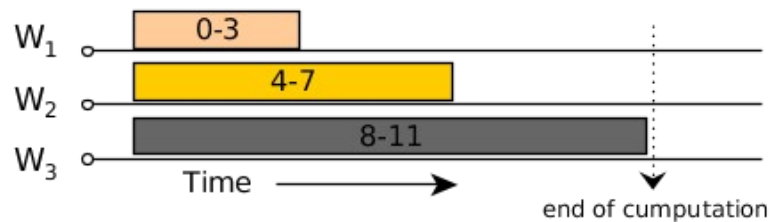
Iterations scheduling: example



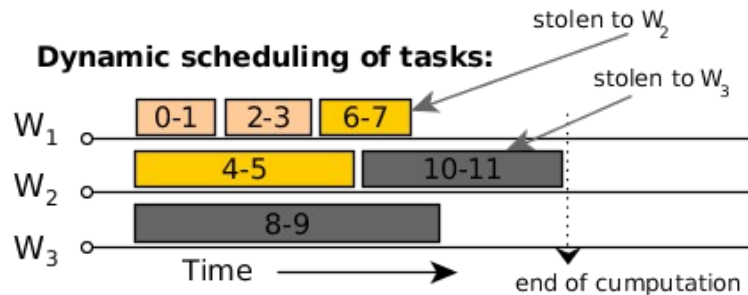
Suppose to have 3 workers and a chunksize=2, then the initial plan used for scheduling iterations is

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

Static assignment of tasks:



Dynamic scheduling of tasks:



Iterations scheduling in the ParallelFor* patterns

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

ParallelForReduce *example*

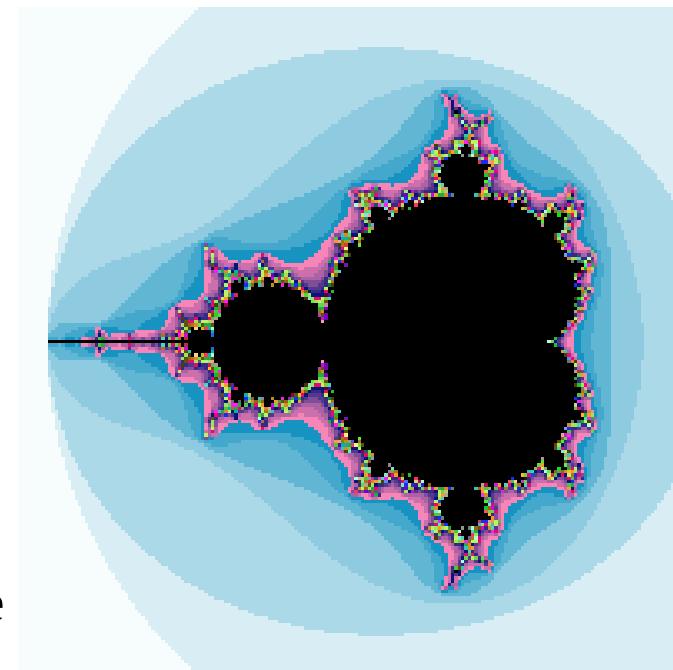
- Dot product (or scalar product or inner product), takes two vectors (A,B) of the same length, it produces in output a single element computed as the sum of the products of the corresponding elements of the two vectors. Example:

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

- Let's comment the FastFlow parallel implementation in the tutorial folder
`<fastflow-dir>/tutorial/fftutorial_source_code/examples/dotprod/dotprod.cpp`

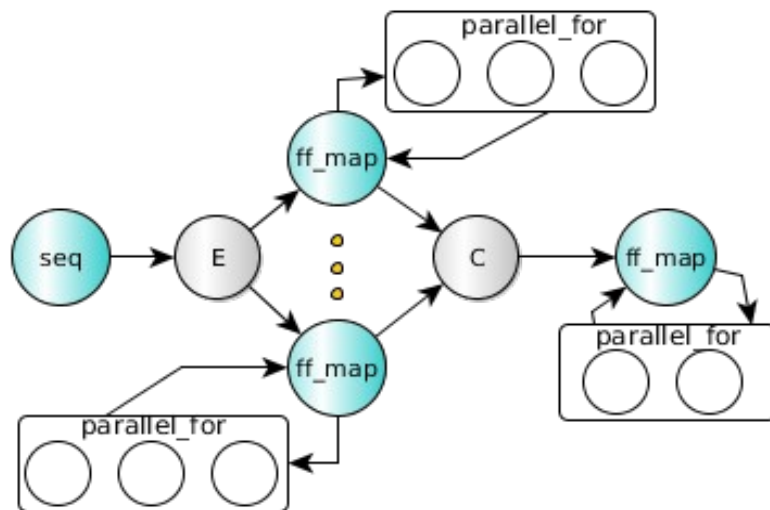
Mandelbrot set example

- Very simple data-parallel computation
 - Each pixel can be computed independently
 - Simple ParallelFor implementation
- Black-pixel requires much more computation
- A naïve partitioning of the images quickly leads to load unbalanced computation and poor performance
 - Let's consider as the minimum computation unit a single image row (image size 2048x2048, max 10^3 iterations per point)
 - ParallelFor Static partitioning of rows (48 threads) chunk=0 **MaxSpeedup ~14**
 - ParallelFor Dynamic partitioning of rows (48 threads) chunk=1 **MaxSpeedup ~37**
 - `<fastflow-dir>/tutorial/fftutorial_source_code/example/mandelbrot_dp/mandel.cpp`



Combining Data Parallel and Stream Parallel Computations

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



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

The ff_Map pattern

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

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

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

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

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

    Task *svc(Task *input) {

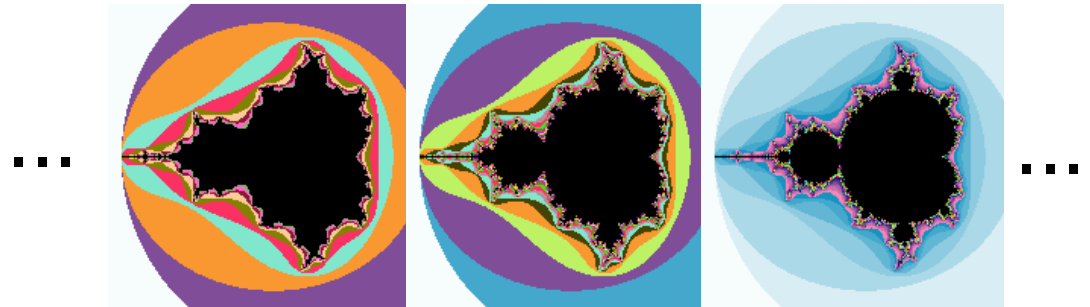
        map::parallel_for(...);

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

        return out;
    }
};
```

Mandelbrot set

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



- We have basically two options:

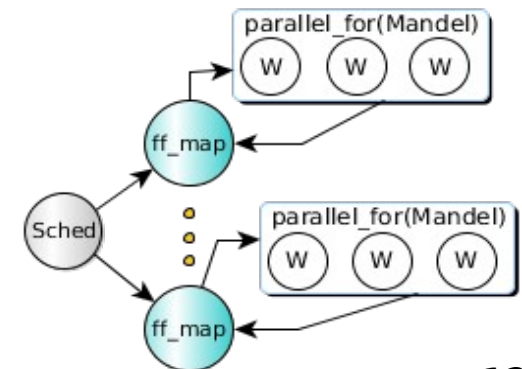
1. One single parallel-for inside a sequential for iterating over all different threshold points
2. A task-farm with map workers implementing two different scheduling strategies

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

- Which one is better having limited resources ?

– Depends on many factors, *too difficult to say in advance*

Moving quickly between the two solutions is the key point

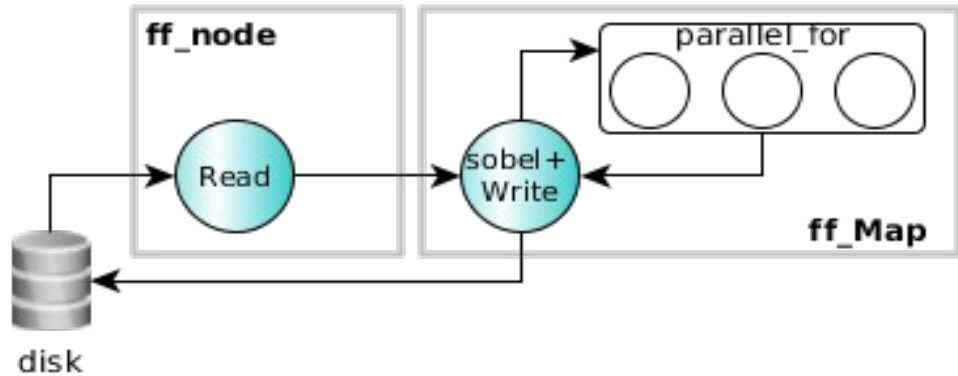




ff_Map *example*

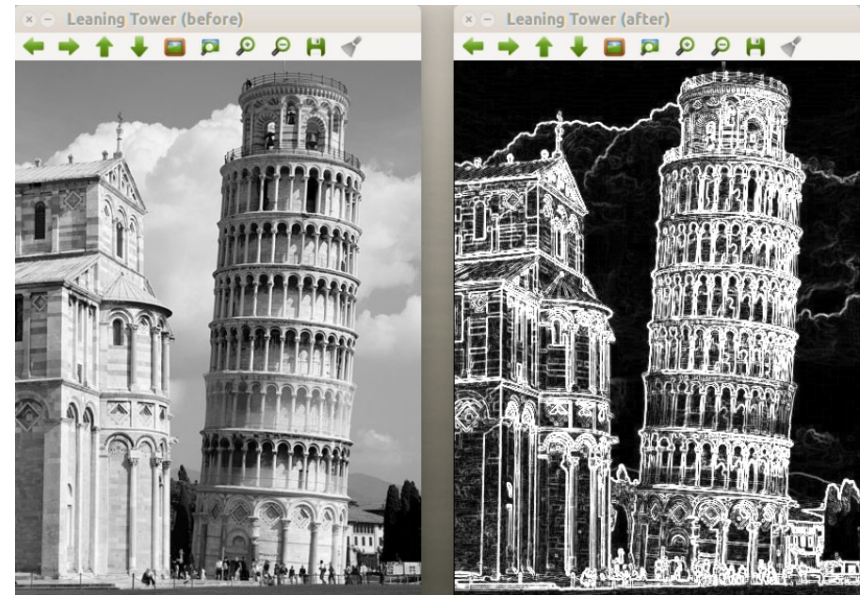
- Let's have a look at the simple test case in the FastFlow tutorial
`<fastflow-dir>/tutorial/fftutorial_source_code/tests/hello_map.cpp`

Parallel Pipeline + Data Parallel : Sobel filter



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

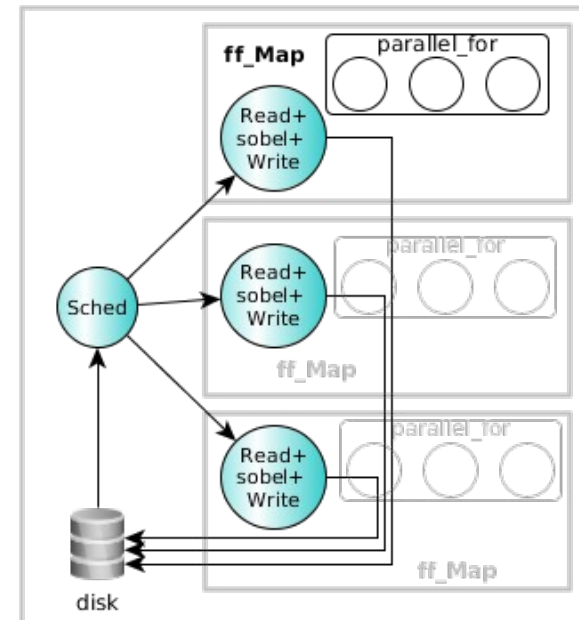
  Task *svc(Task*task) {
    Mat src = *task->src, dst= *task->dst;
    ff_Map<>::parallel_for(1,src,src.row-1,
      [src,&dst](const long y) {
        for(long x=1;x<src.cols-1;++x) {
          .....
          dst.at<x,y> = sum;
        }
      });
    const std::string outfile="./out"+task->name;
 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



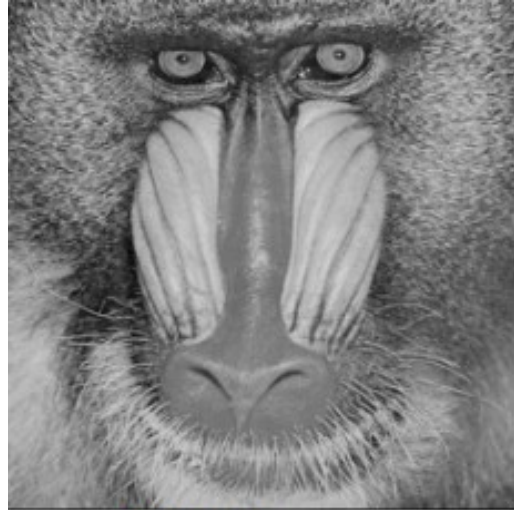
- 2 Intel Xeon CPUs E5-2695 v2 @ 2.40GHz (12x2 cores)
- 320 images of different size (from few kilos to some MB)
- sobel (seq): ~ 1m
- pipe+map (4): ~15s
- farm+map (8,4): ~5s
- farm+map (32,1): ~3s

Two stage image restoration

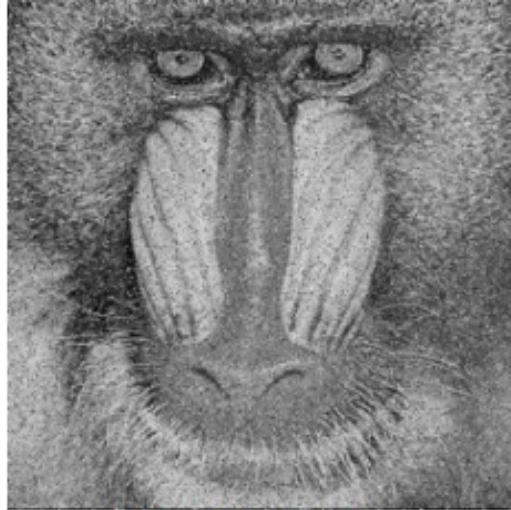


- Detect: adaptive median filter, produces a noise map
- Denoise: variational Restoration (iterative optimization algorithm)
 - 9-point stencil computation
- High-quality edge preserving filtering
- Higher computational costs w.r.t. other edge preserving filters
 - without parallelization, no practical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

Two stage image restoration: Salt & Pepper image



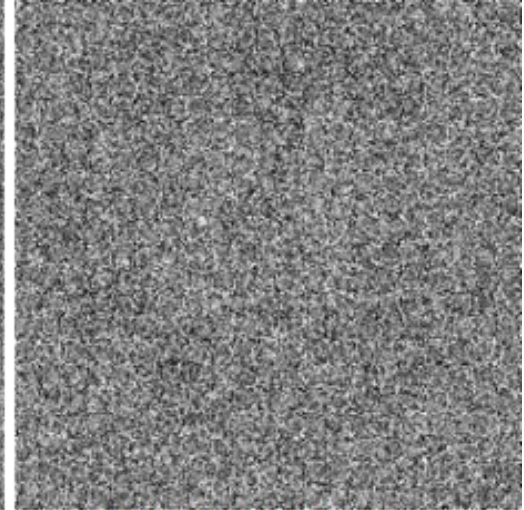
Original
Baboon
standard
test image
1024x1024



10% impulsive
noise

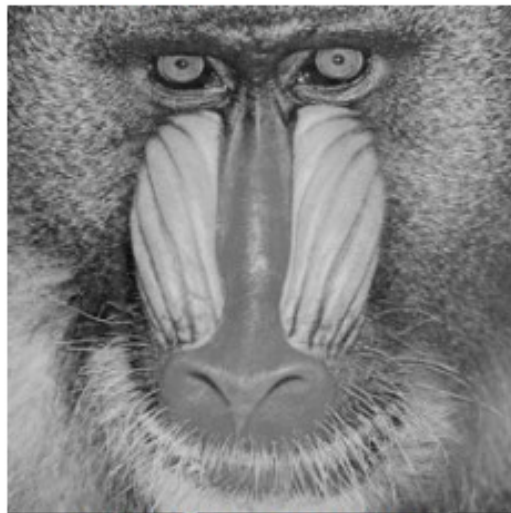


50% impulsive
noise

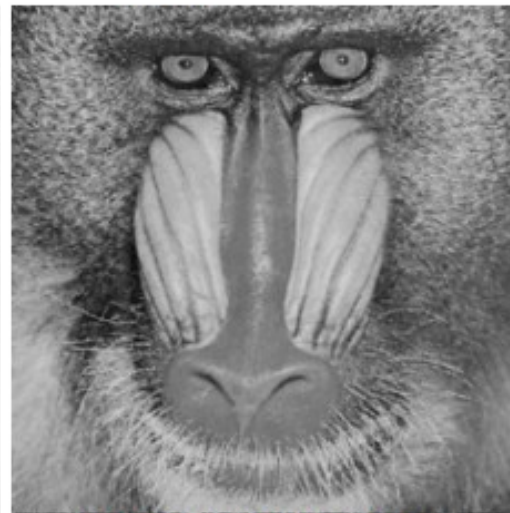


90% impulsive
noise

Restored



PNSR 43.29dB MAE
0.35



PNSR 32.75dB MAE
2.67



PNSR 23.4 MAE
11.21