Introduction to FastFlow programming

Massimo Torquati
<massimo.torquati@unipi.it>
SPM lecture 3

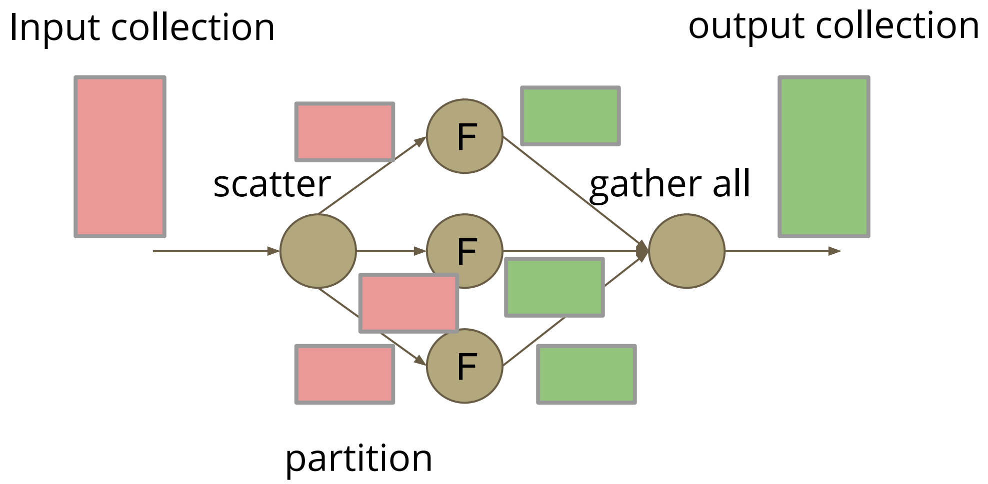
Master Degree in Computer Science
Master Degree in Computer Science & Networking
University of Pisa

Data Parallel (DP) Computations (recap)

- Large collections of data are partitioned among the number of computing resources (threads, processes, nodes, etc.) each one executing the same function F over the assigned partition of the collection
- The computation may be inplace (typical in *map-based* computations)
- Usually the so-called “owner computes rule” is applied (typical in *stencil-based* computations)
    - On shared-memory systems, the function F has read/write access to the elements of the assigned partition and read-only access to (some) other elements of the input collection
- The main goal of data parallel computations is to reduce the *completion time* to compute the entire collection
- Usually they are encountered in sequential programs as loop-based computations

Data Parallel (DP) Computations (recap)

- Logical implementation of a DP computation



Data Parallel Computations in FastFlow

- Data parallel patterns are: Map, Reduce, Map+Reduce, Stencil, Stencil+Reduce
- In FastFlow all DP patterns have been implemented by using the ParallelFor/ParallelForReduce patterns
  - The Map pattern in FastFlow is based on the ParallelForReduce. It implements a pure Map and a Map+Redure
  - We will see that the Map pattern can be used as a Worker of a farm and/or as a stage of a pipeline.
  - We will not consider the StencilReduce pattern

ParallelFor

- The *ParallelFor* pattern can be used to parallelize loops with independent iterations
- The class interface is defined in the file *ff/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(8);   // defining the object

pf.parallel_for(0, N, 1, 0, [&A,B](const long i) {
     A[i] = A[i] + B[i];
}, 4);
```

- Constructor interface (all arguments have a default value)
  - *ParallelFor(maxnworkers, spinWait, spinBarrier);*
- On the basis of the number and types of arguments you have different *parallel_for* signatures
  - *parallel_for(first-index, last-index, stepsize, chunksize, bodyFunction, nworkers);*
  - The *bodyFunction* is a C++ lambda

Example: sum of the square

- Previously implemented generating a stream of N floats and using a pipeline+farm schema
- Basically we unfolded the loop:

```
sum=0.0;
for(float i=0.0; i<N; i++) sum += i*i;
```

- Let's use now a ParallelFor for computing the square of the elements of a std::vector<float>
- The reduction (i.e. the total sum) will be computed sequentially
  - See the pf_square.cpp file

ParallelForReduce

- The *ParallelForReduce* pattern can be used to parallelize loops with independent iterations and having reduction variables (map+reduce pattern)
- Example:

```
// A is an array of long integers 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,1,0, [](const long i, long &mysum) {
           mysum += A[i] + B[i];
   },
   [ ](long &s, const long e) { s += e;}
);
```

- Constructor interface is the same of the ParallelFor (but the template type)
- *parallel_ruduce* method interface:
    - *parallel_reduce(var, identity-val, first, last, step, chunksize, mapF, reduceF, nworkers);*
    - *mapF* and *reduceF* are C++ lambdas

Example: sum of the square

- Let’s use a ParallelForReduce for computing the square and the reduction
  - No array needed we can parallelize directly the initial loop

```
sum=0.0;
for(float i=0.0; i<N; i++) sum += i*i;
```

- See pf_square2.cpp

Example: dot product

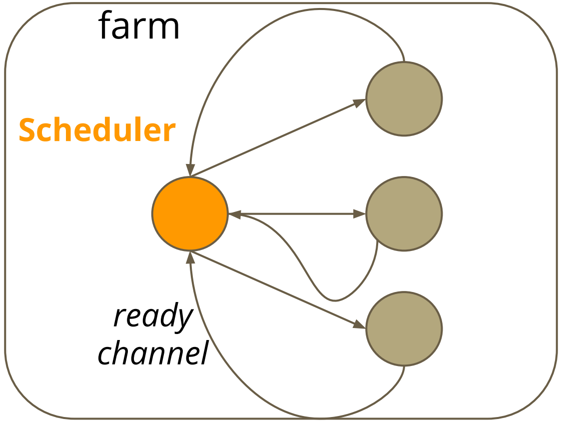
- The dot product operation (or scalar product, inner product) takes two vectors (A, B) of the same length, it produces in output a single element that is the sum of the products of the corresponding elements of the two vectors. Example:

```
std::vector<double> A(N), B(N);
..... // A and B initialized
double s=0.0;
for(size_t i=0; i<N; i++) s += A[i]*B[i];
```

- Let's have a look at the test_dotprod_parfor.cpp inside the tests directory

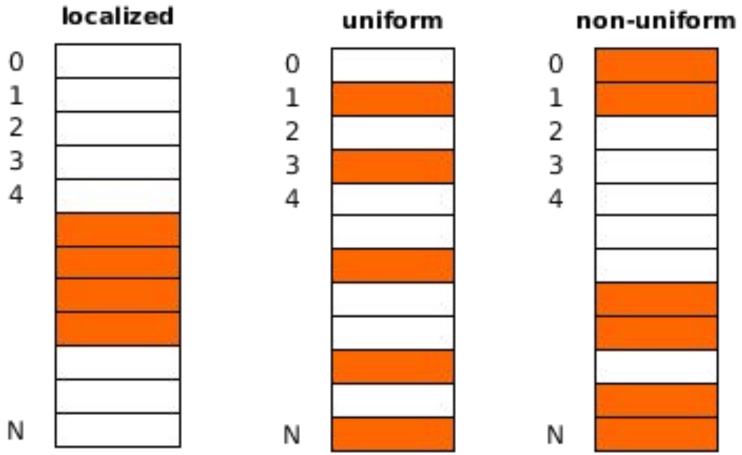
ParallelFor* implementation

- The ParallelFor* patterns are implemented on top of the farm building block
- The skeleton is a master-worker, where the master is the Scheduler of loop iterations
- The Scheduler can be disabled by calling the ParallelFor* method *disableScheduler()*



Iterations scheduling

- Let's consider the following case
  - for(size_t i=0;i<N;++i) A[i] = F(A[i]);   //map-like computation
- The time difference for computing distinct elements in the array A may be large (or very large)
- Problem: How do we schedule the loop's iterations?

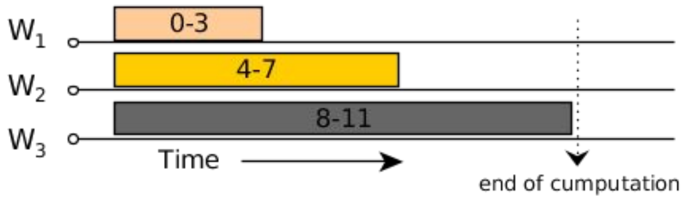


Iterations scheduling

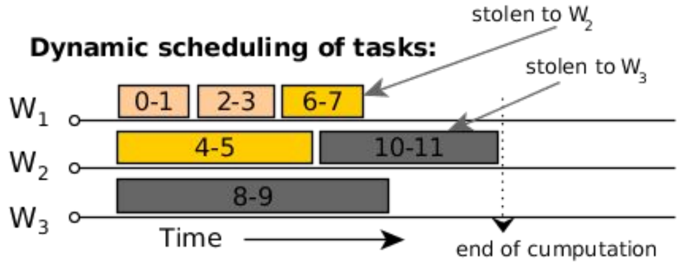
Suppose to have 3 workers and a chunksize=2, then the initial plan used for scheduling iterations is

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

Static assignment of tasks:



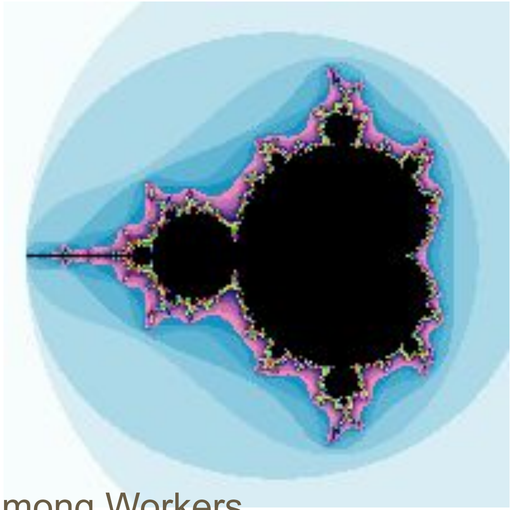
Dynamic scheduling of tasks:



Iterations scheduling in the ParallelFor* patterns

- Iterations are scheduler according to the value of the *chucksize* parameter
  - *parallel_for(first-index, last-index, stepsize,* ***chunksize****, bodyFunction, nworkers);*
  - *parallel_reduce(var, identity-val, first, last, step,* ***chunksize****, mapF, reduceF, nworkers);*
- Three options:
  1. **chunksize = 0** : static scheduling
     - Each worker gets a contiguous chunk of ~(#iteration / #workers) iterations
  2. **chunksize > 0** : dynamic scheduling with task granularity equal to *chunksize*
  3. **chunksize < 0** : static scheduling with task granularity equal to *chunksize*, chunks are assigned to workers in a round-robin fashion

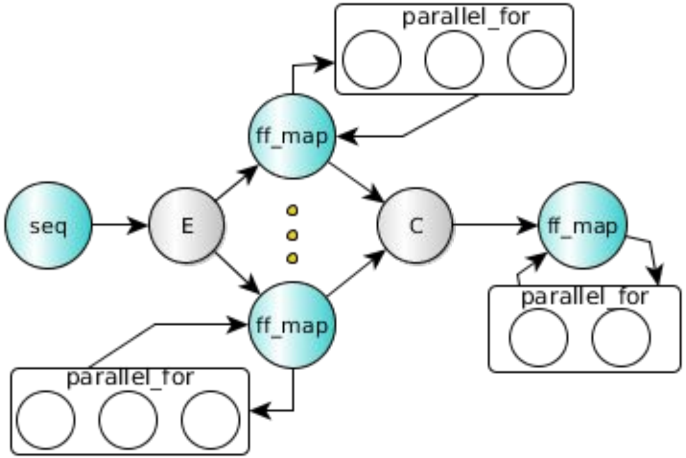
Mandelbrot set example



- Simple DP computation:
  - Each pixel of the image can be computed independently
  - Black pixels require much more iterations
  - Easy to implement by using a farm or a ParallelFor
- A static partitioning of the image quickly leads to unbalanced computation among Workers
- Example:
  - Image size 2048x2048, max iterations per point 10^3, 48 Workers, 24 cores (2-way HT)
  - Static partitioning of rows (i.e. chunksize=0) **Max Speedup ~14**
  - Dynamic partitioning of rows (i.e. chunksize=1) **Max Speedup ~37**
- Code: mandel.cpp

Combining DP and Stream Parallel computations

- If one of the stages of a pipeline is a bottleneck its service time can be reduced by parallelizing the stage
- If the stage computes a Map or a Map+Reduce it can be parallelized by using a ParallelFor* pattern



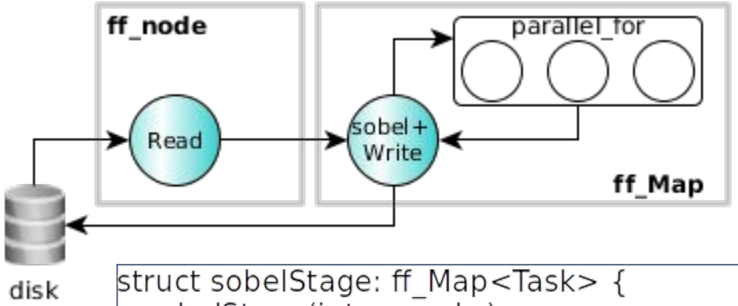
- Two options:
  - Using a ParallelFor* pattern inside the svc method
  - Replacing the node with a ff_Map pattern

The FastFlow Map Pattern

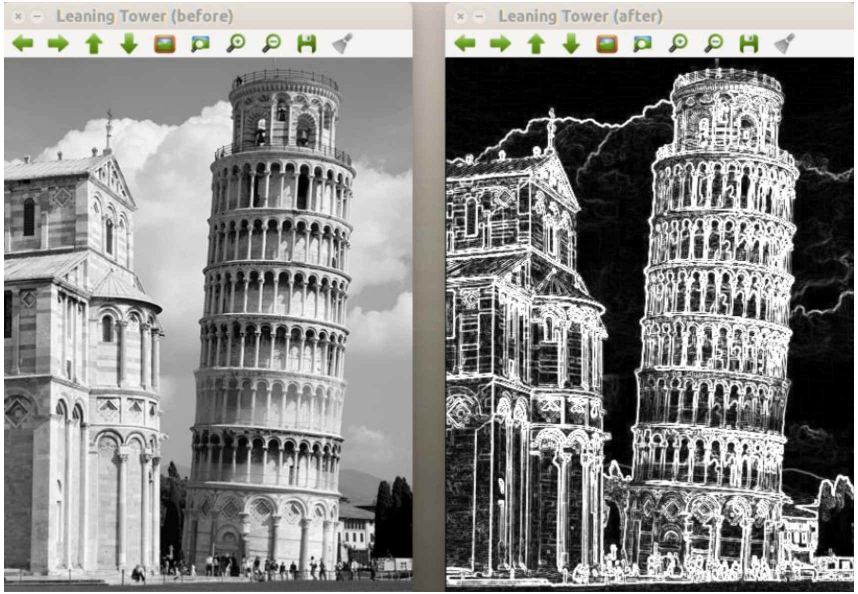
- The FastFlow Map Pattern (ff_Map) is just a single-input single-output node that wraps a ParallelForReduce pattern
  - *ff_Map<IN_t, OUT_t, reduce-variable-type>*
- Inside a pipeline or a farm, it is generally better to use the ff_Map than a plain ParallelForReduce because some optimizations are automatically introduced by the Map (mapping of worker threads, scheduler disabled, etc..)

```
#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;
    }
};
```

Example: Image filtering



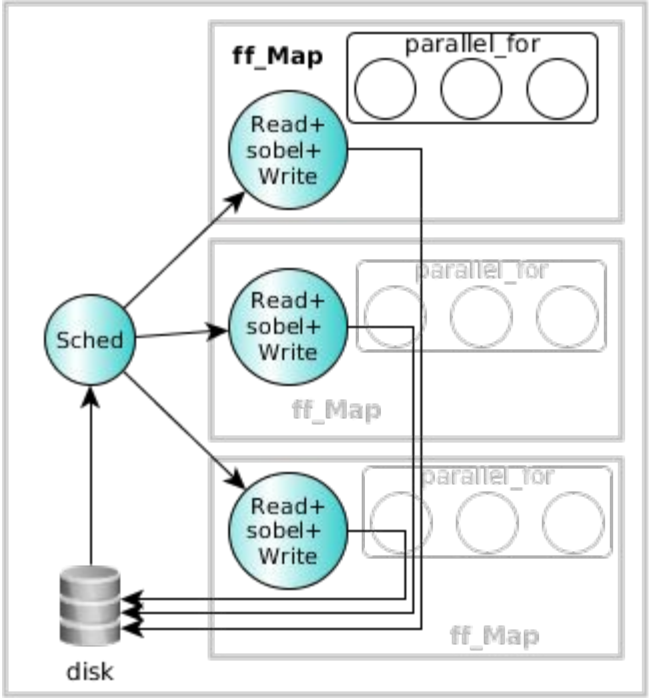

```
struct sobelStage: ff_Map<Task> {
  sobelStage(int mapwks):
     ff_Map<Task>(mapwrks, true) {};

   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 some image from the disk
- Each image is converted in B&W and sent to the next stage
- The second stage applies the ***Sobel filter*** to each image by using OpenCV and then writes the result into the disk

Example: Image filtering

- We can parallelize the second stage by using a farm because the image are independent
- We can also normalize the farm (see the figure)
    - The emitter schedules file names of the image using an on-demand policy,
    - the Worker reads the image in parallel, then applies the filter using a Map and finally writes the resulting image on the disk (with a different name)
- There are two levels of parallelism: th n. of farms Workers, and the n. of Map Workers, how to balance them?



Example: Sobel filter

- Machine: 2 CPUs Xeon E5-2695 @2.4GHz eche CPU has 12 cores 2-way HyperThreading
- 320 images of different size (from a few KBs to some MBs)
- **Results:**
  - **a)** sequential ~1m;
  - **b)** pipe(seq, map(4)) ~15s;
  - **c)** farm(map(4), 8) ~5s;
  - **d)** farm(seq, 32) ~3s
- The d) version is the *normal form*, it provides always the best performance (if it can be used)