Introduction to FastFlow programming

SPM lecture, November 2015

Massimo Torquati <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy

ClassWork5: comments

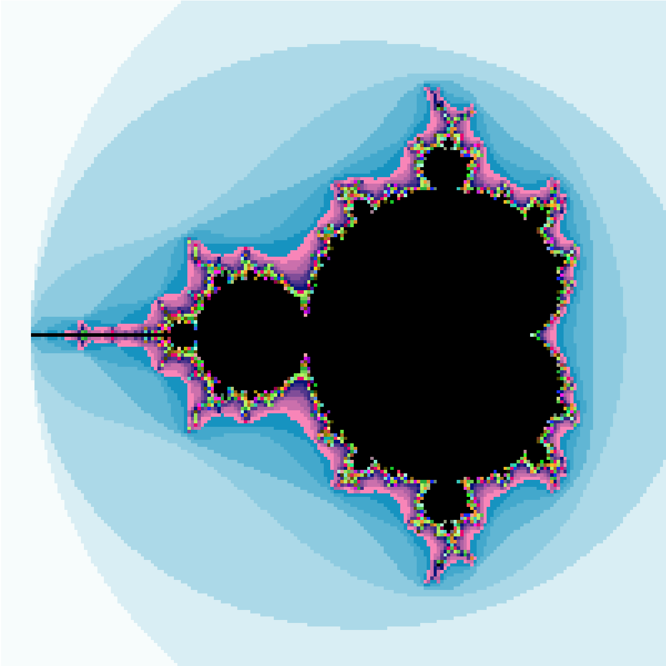
- Let's comment on a possible solution for the ClassWork4. Take a look in the ClassWork5 folder:

~spm1501/public/ClassWork45/primes_parallelfor.cpp

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 ~(#iteration-space/#workers) iterations
  - chunk > 0: dynamic scheduling with task granularity equal to chunk iterations
  - chunk < 0: static scheduling with task granularity equal to chunk, chunks are assigned to workers in a round-robin fashion

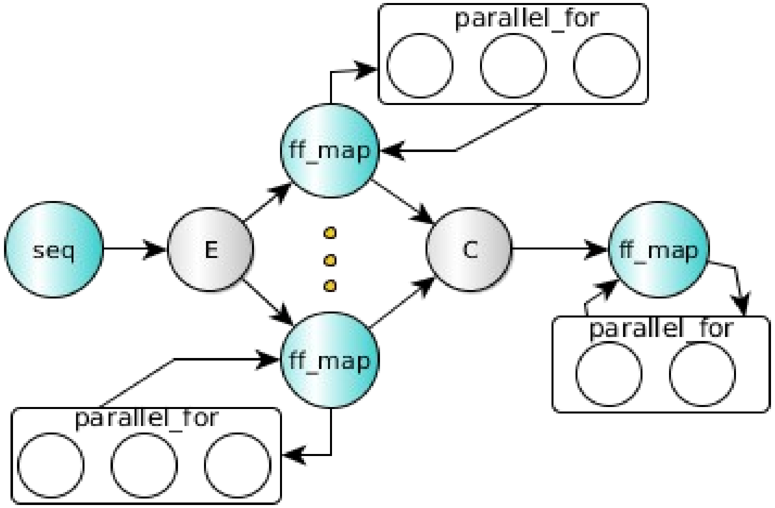
Mandelbrot set example



- 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 the minimum computation unit a single image row (image size 2048x2048, max 10^3 iterations per point)
    - ParallelFor Static partitioning of rows (48 threads) **MaxSpedup 14**
    - ParallelFor Dynamic partitioning of rows (48 threads) **MaxSpeedup 37**

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;
    }
};
```

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

ClassWork6

- Consider the following case:
  - In input we have a stream of *k* matrices of size NxM. Let *S* be a vector representing an internal state having size M.
  - For each input matrix the program computes
    - T = M*S (matrix vector product)
    - s = sum T[i] (getting the sum of all elements, reduce operation)
    - S[i] += s (updating the internal state with the result of the reduce)
  - At the end of the data stream, the result produced is s = sum S[i]
- Give a parallel implementation of the problem by using the FastFlow pipeline and ff_Map. The first stage of the pipeline produces the k matrices.