Supporting Advanced Patterns in GrPPI, a Generic Parallel Pattern Interface

David del Rio, Manuel F. Dolz, Javier Fernández. Daniel Garcia

University Carlos III of Madrid

Autonomic Solutions for Parallel and Distributed Data Stream Processing (Auto-DaSP) – Euro-Par 2017

Santiago de Compostela, August 29th, 2017

RePhrase Project: Refactoring Parallel Heterogeneous Software – a Software Engineering Approach

(ICT-644235), 2015-2018, €3.6M budget

8 Partners, 6 European countries UK, Spain, Italy, Austria, Hungary, Israel

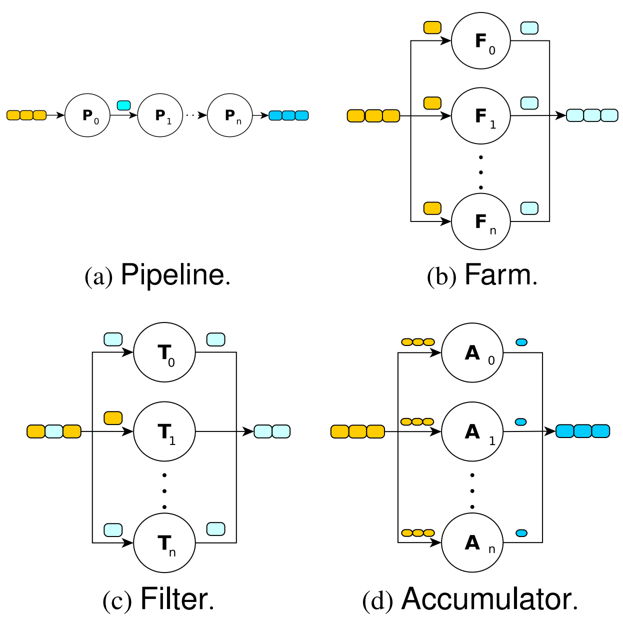
http://www.rephrase-ict.eu

Thinking in Parallel

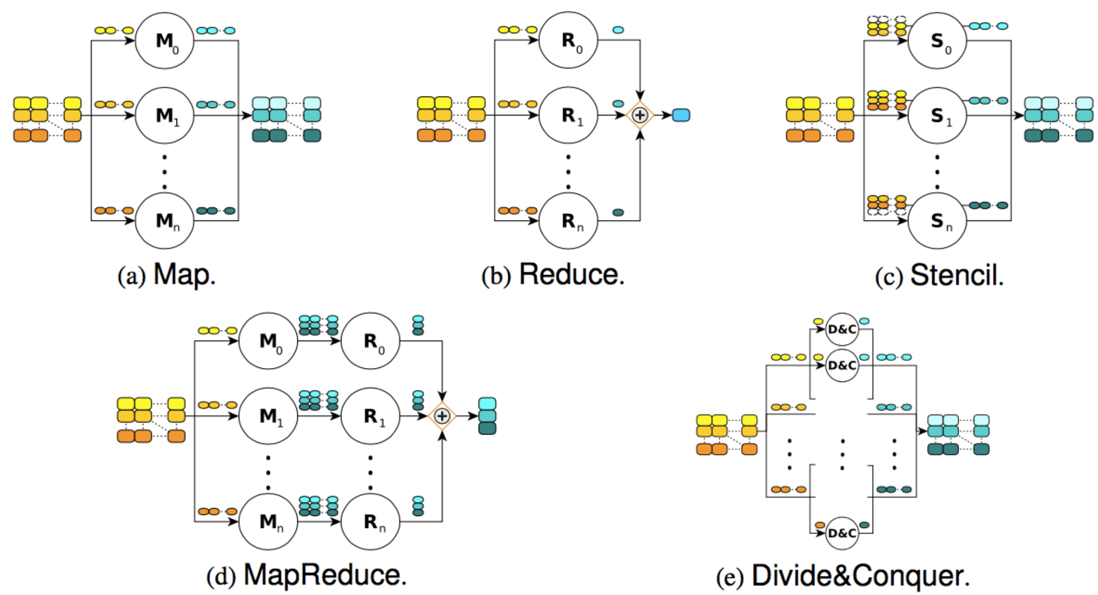
- Fundamentally, programmers must learn to ***"think parallel"***
  - this requires new *high-level* programming constructs
  - you cannot program effectively while worrying about deadlocks etc.
    - they must be eliminated from the design!
  - you cannot program effectively while handling with communication etc.
    - this needs to be packaged/abstracted!
  - you cannot program effectively without performance information
    - this needs to be included!
- **How this can be solved? We use two key technologies:**
  - Refactoring (changing the source code structure)
  - Parallel Patterns (high-level functions of parallel algorithms)

GrPPI: A Generic and Reusable Parallel Pattern Interface

- **Objectives:**
  - Provide a high-level parallel pattern interface for C++ applications
  - ***A Generic and Reusable Parallel Pattern Interface (GrPPI)***
    - High-level C++ interface based on metaprogramming and template-based techniques
    - Support for different parallel programming frameworks as back ends
    - Support for basic stream and data patterns and their composability



(a) Pipeline. (b) Farm. (c) Filter. (d) Accumulator.



(a) Map. (b) Reduce. (c) Stencil. (d) MapReduce. (e) Divide&Conquer.

GRPPI: A Generic and Reusable Parallel Pattern Interface

- **Example of the *Pipeline* pattern interface:**

Listing 1.1: Pipeline interface.

```
template <typename ExecMod, typename InFunc, typename ... Arguments>
void Pipeline( ExecMod m, InFunc in, Arguments ... sts );
```

Listing 1.2: Usage example of the Pipeline pattern.

```
Pipeline( parallel_execution<OMP>,
    // Stage 0: read values from a file
    [&]() {
        auto r = read_list(is);
        return ( r.size() == 0 ) ? optional<vector<int>>{} : make_optional(r);
    },
    // Stage 1: takes the maximum value of the vector
    [&]( optional<vector<int>> v ) {
        return ( v->size() > 0 ) ?
            make_optional( *max_element(begin(*v), end(*v)) ) :
            make_optional( numeric_limits<int>::min() );
    },
    // Stage 2: prints out the result
    [&os]( optional<int> x ) {
        if (x) os << *x << endl;
    }
);
```

Journal publication: D. R. Astorga, M. F. Dolz, J. Fernandez and J. D. Garcia, "A generic parallel pattern interface for stream and data processing," Concurrency and Computation: Practice and Experience, http://dx.doi.org/10.1002/cpe.4175. OpenAccess available.

https://github.com/arcosuc3m/grppi

GRPPI: A Generic and Reusable Parallel Pattern Interface

- **Patterns vs. Parallel programming frameworks:**
  - **Stream and data parallel patterns – They can be composed among them!**

GRPPI

Basic patterns

Stream patterns	Sequential	OpenMP	TBB	C++ Threads	CUDA Thrust
Pipeline	✓	✓	✓	✓	✗
Farm	✓	✓	✓	✓	✓
StreamFilter	✓	✓	✓	✓	✗
StreamAcumulator	✓	✓	✓	✓	✓
StreamIterator	✓	✓	✓	✓	✗
Data patterns	Sequential	OpenMP	TBB	C++ Threads	CUDA Thrust
Map	✓	✓	✓	✓	✓
Stencil	✓	✓	✓	✓	✗
Reduce	✓	✓	✓	✓	✓
MapReduce	✓	✓	✓	✓	✓
Divide&Conquer	✓	✓	✓	✓	✗

Journal publication: D. R. Astorga, M. F. Dolz, J. Fernandez and J. D. Garcia, "A generic parallel pattern interface for stream and data processing," Concurrency and Computation: Practice and Experience, http://dx.doi.org/10.1002/cpe.4175. OpenAccess available.

https://github.com/arcosuc3m/grppi

GRPPI: A Generic and Reusable Parallel Pattern Interface

- **Patterns vs. Parallel programming frameworks:**
  - **Stream and data parallel patterns – They can be composed among them!**
  - **We include advanced patterns!**

Basic patterns

Stream patterns	Sequential	OpenMP	TBB	C++ Threads	CUDA Thrust
Pipeline	✓	✓	✓	✓	✗
Farm	✓	✓	✓	✓	✓
StreamFilter	✓	✓	✓	✓	✗
StreamAcumulator	✓	✓	✓	✓	✓
StreamIterator	✓	✓	✓	✓	✗
Data patterns	Sequential	OpenMP	TBB	C++ Threads	CUDA Thrust
Map	✓	✓	✓	✓	✓
Stencil	✓	✓	✓	✓	✗
Reduce	✓	✓	✓	✓	✓
MapReduce	✓	✓	✓	✗	✓
Divide&Conquer	✓	✓	✓	✓	✗

Advanced patterns

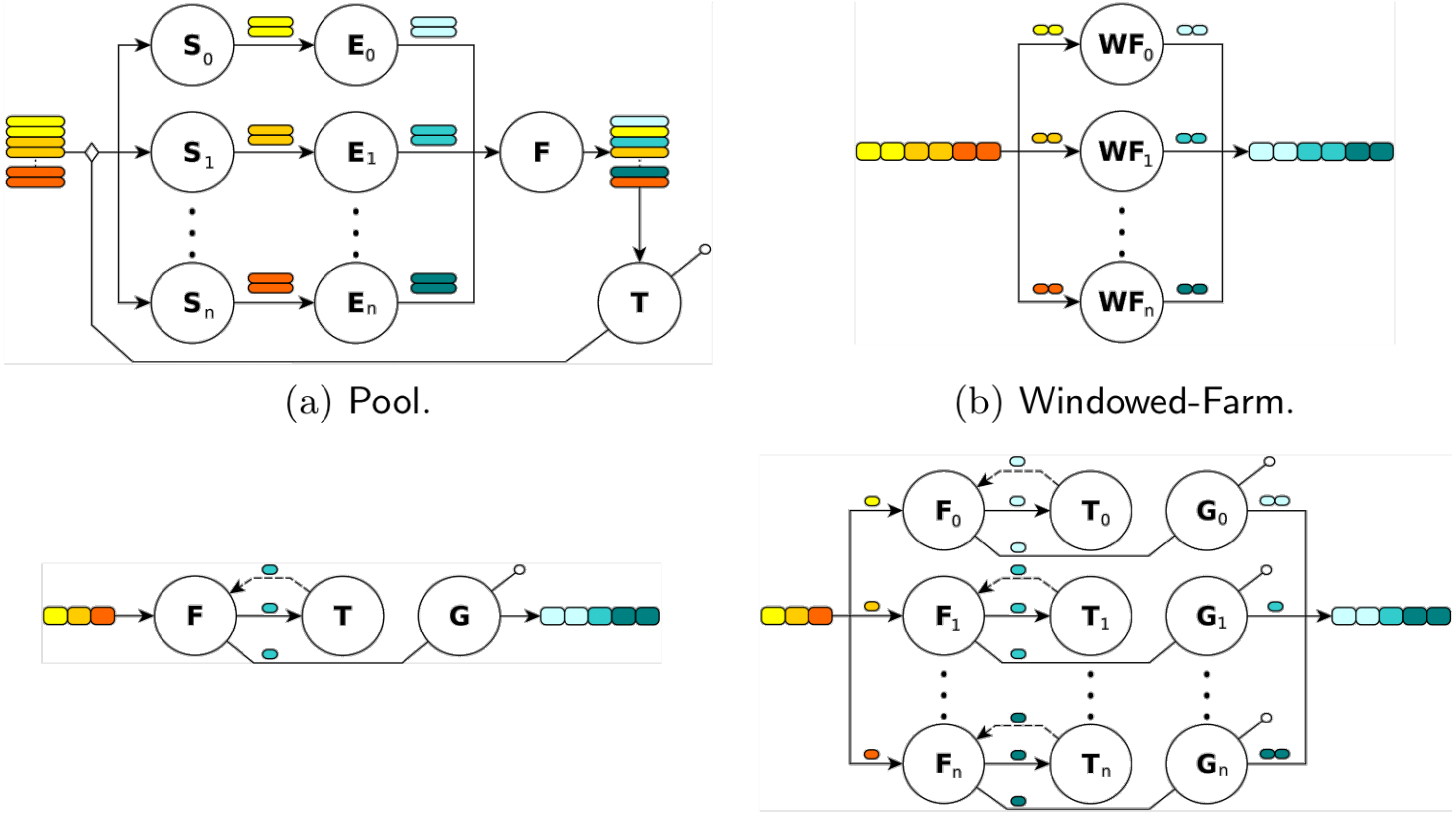
Stream/data	Sequential	OpenMP	TBB	C++ Threads	CUDA Thrust
???	***???***	***???***	***???***	***???***	***???***

Journal publication: D. R. Astorga, M. F. Dolz, J. Fernandez and J. D. Garcia, “A generic parallel pattern interface for stream and data processing,” Concurrency and Computation: Practice and Experience, http://dx.doi.org/10.1002/cpe.4175. OpenAccess available.

https://github.com/arcosuc3m/grppi

Advanced parallel patterns

- New advanced patterns in GrPPI: **Pool, Windowed-Farm and Stream-Iterator**

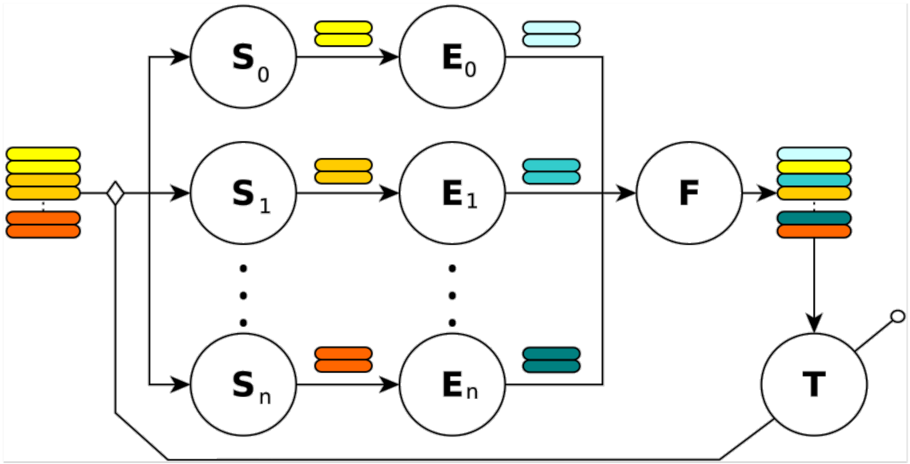


(a) Pool. (b) Windowed-Farm.

(c) Stream-Iterator. (d) Farm–Stream-Iterator.

Advanced parallel patterns

- The Pool pattern:



S=Selection; E=Evolution; F=Filter; T=Termination

- Models the evolution of a population of individuals
- Applies iteratively the following functions on the population P:
  - **Selection (S):** Selects sepecific individuals (pure func.)
  - **Evolve (E):** Evolves individuals to any number of new or modified individuals (pure func.)
  - **Filter (F):** Filters individuals and inserts them back into the population
  - **Termination (T):** Determines whether the evolution should finish or continue

Advanced parallel patterns

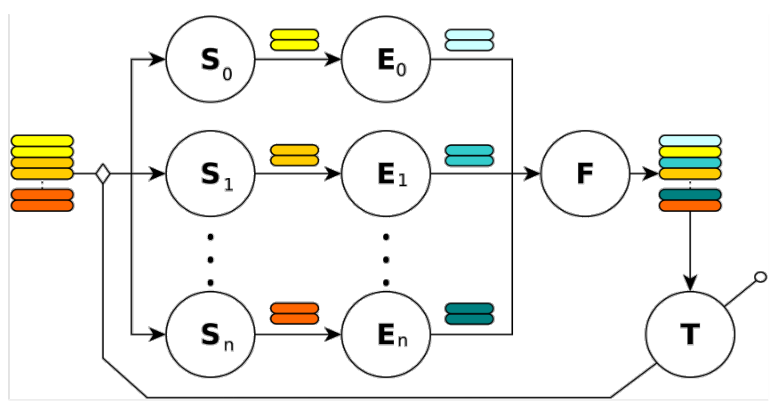
- The <u>Pool pattern</u> is mainly used in:
  - ***Symbolic computing domain***: Orbit patterns.
  - ***Evolutionary computing domain***: Genetic algorithm patterns, multiagent systems, etc.
  - ***Example***: Travelling salesman, a NP-problem computing the shortest path among cities.

- GrPPI interface:

Listing 1.1: Pool interface.

```
template <typename EM, typename P, typename S, typename E, typename F, typename T>
void Pool(EM exec_mod, P &popul, S &&select, E &&evolve, F &&filt, T &&term, int num_select);
```

Parallelism degree
(`EM exec_mod`)

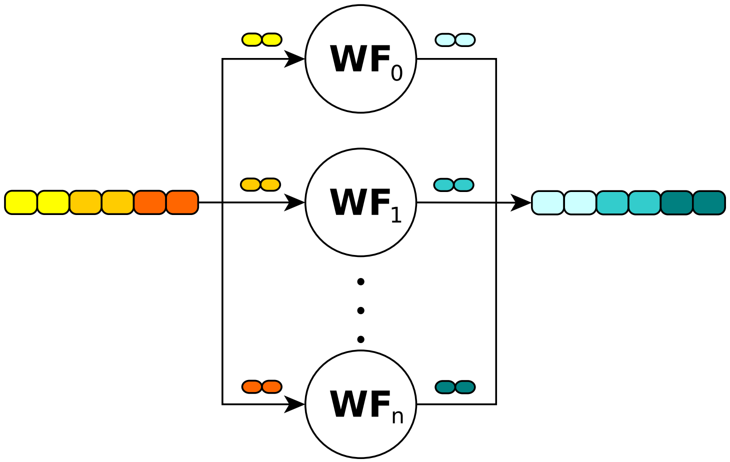


Number of selections
(`int num_select`)

(a) Pool.

Advanced parallel patterns

- The Windowed-Farm pattern:



- Stream pattern that delivers windows of processed items to the output stream
- Performs the following actions:
  - The (pure) function window-farm **WF** transforms consecutive windows of size x to windows of size y.
  - The output items may contain the items from the input windows collapsed in a specific way.
  - The windows can have an overlap factor.

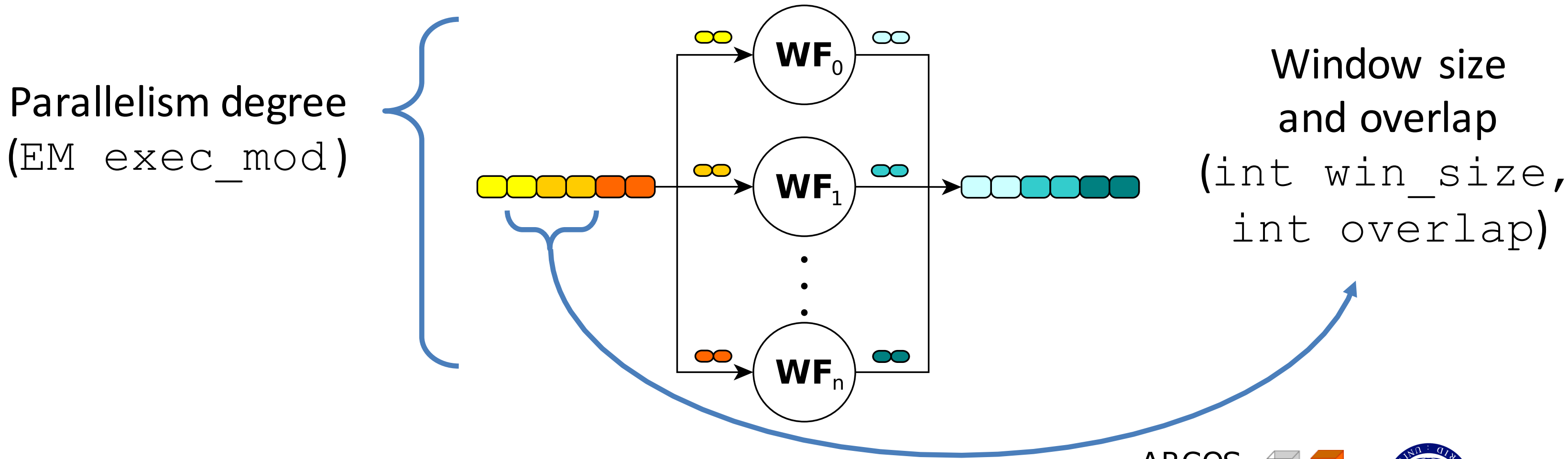
Advanced parallel patterns

- The Windowed-Farm pattern can be used in:
  - ***Real-time processing engines.***
  - ***Wireless sensor networks.***
  - ***Example***: compute average window values from sensor readings.

- GrPPI interface

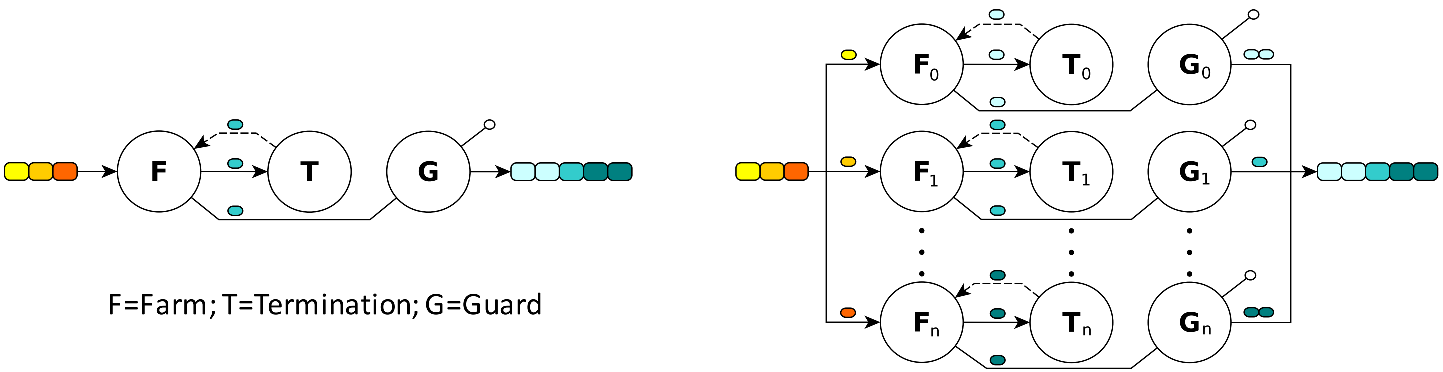
Listing 1.2: Windowed-Farm interface.

```
template <typename EM, typename I, typename WF, typename O>
void WindowedFarm(EM exec_mod, I &&in, WF &&task, O &&out, int win_size, int overlap);
```



Advanced parallel patterns

- The *stand-alone* and *farmed* Stream-Iterator pattern:



(c) Stream-Iterator. (d) Farm–Stream-Iterator.

- Stream pattern that recurrently computes a given pure function
- Applies the following functions on the input stream items:
  - **Farm (F):** transforms a single stream input; it can be computed in parallel (pure func.).
  - **Termination (T):** determines whether the computation of F should be continued or not.
  - **Guard (G):** determines in each iteration if the result of the function F should delivered to the output stream or not.

Advanced parallel patterns

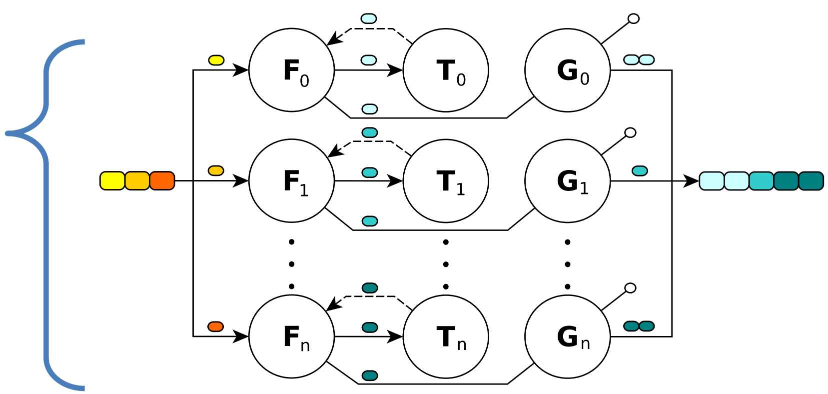
- The *stand-alone* and *farmed* Stream-Iterator patterns:
  - ***Real-time processing applications***
  - ***Example***: reduce to different resolutions the frames appearing on an input video. The resolution is halved in each iteration.

- GrPPI interface

Listing 1.3: Stream-Iterator interface.

```
template <typename EM, typename I, typename F, typename O, typename T, typename G>
void StreamIteration(EM exec_mod, I &&in, F &&task, O &&out, T &&term, G &&guard);
```

Parallelism degree (`EM exec_mod`)



Advanced parallel patterns

- *Composability features: Stream-Iterator + Pipeline*

```
StreamIteration( parallel_execution_thr{4},
    [&]() -> optional<int> { // Consumer function
        auto value = read_value(is);
        return ( value > 0 ) ? value : {};
    },
    Pipeline( // Kernel function
        []( int e ) { return e + 2*e; },
        []( int e ) { return e - 1; }
    ),
    // Producer function
    [&]( int e ){ os << e << endl; },
    // Termination function
    [] ( int e ){ return e < 100; },
    // Output guard function
    [] ( int e ){ return e % 2 == 0; }
);
```

Experimental evaluation

- **Usability** and **performance** evaluation of the parallel patterns:
  - ***Target platform***: 2x Intel Xeon Ivy Bridge E5-2695 (24 cores)
  - ***Parallel technologies***: C++11 threads, OpenMP and Intel TBB
  - ***Benchmarks:***
    - **Pool pattern:** travelling salesman (TSP) using a regular evolutionary algorithm. NP-problem computing the shortest route among different cities, visiting them only once and returning to the origin city.
    - **Window-Farm pattern**: computation of average window values from an emulated sensor readings.
    - **Stream-Iteration pattern**: reduction of the resolution of the images appearing in the input stream , and producing images of different resolutions to the output stream.

Experimental evaluation

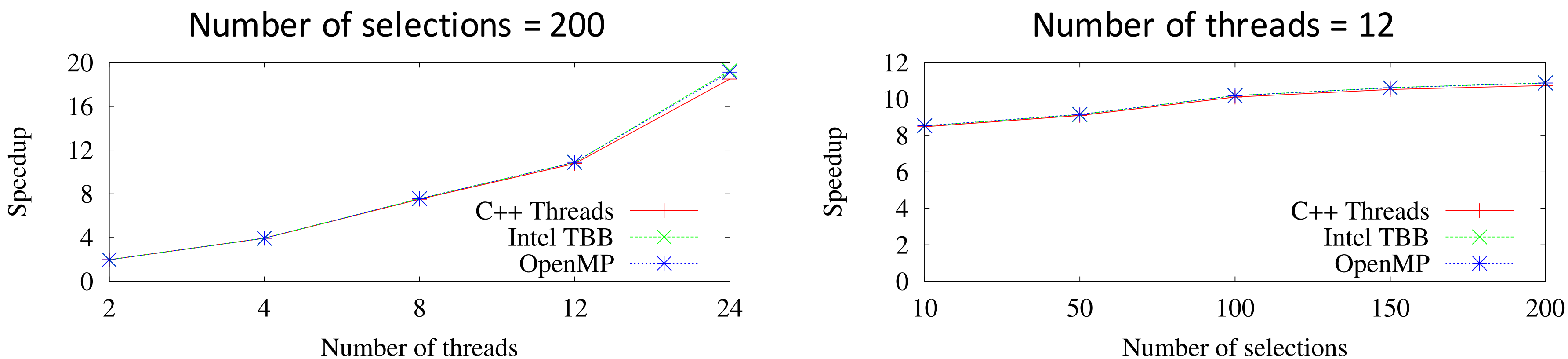
- **Usability performance**
  - Analysis of the modified lines of code (LOCs) w.r.t the sequential version

Advanced pattern	% of modified lines of code			
	C++ Threads	**OpenMP**	**Intel TBB**	**GrPPI**
Pool	+55.0 %	+70.0 %	+55.0 %	+22.5 %
Windowed-Farm	+152.1 %	+75.8 %	+51.7 %	+31.0 %
Stream-Iterator	+153.5 %	+56.4 %	+46.1 %	+30.8 %

  - C++ threads requires more modified LOCs than other frameworks providing high-level interfaces (OpenMP and Intel TBB)
  - Windowed-Farm and Stream-Iterator are more difficult to parallelize!
  - GrPPI requires less modified LOCs than any other framework

Experimental evaluation

- **Pool pattern** pattern evaluation using the travelling salesman problem:
  - Population of 50 individuals representing feasible routes



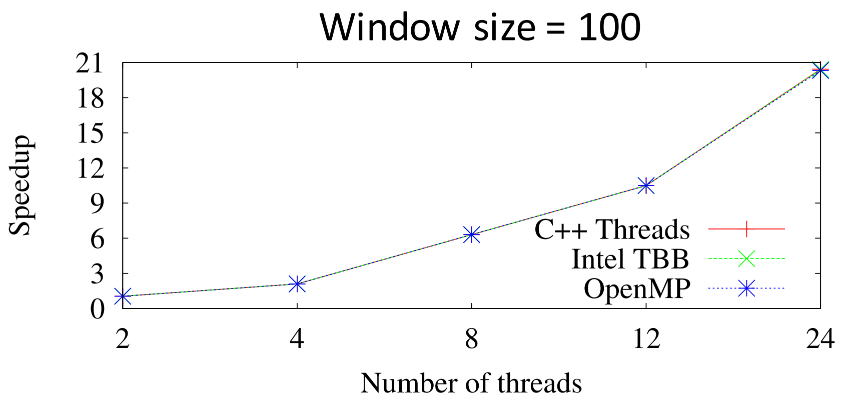
(a) Speedup vs. number of threads. (b) Speedup vs. number of selections.

a) Good speedup scaling w.r.t number of threads, however Intel TBB and OpenMP back ends perform better

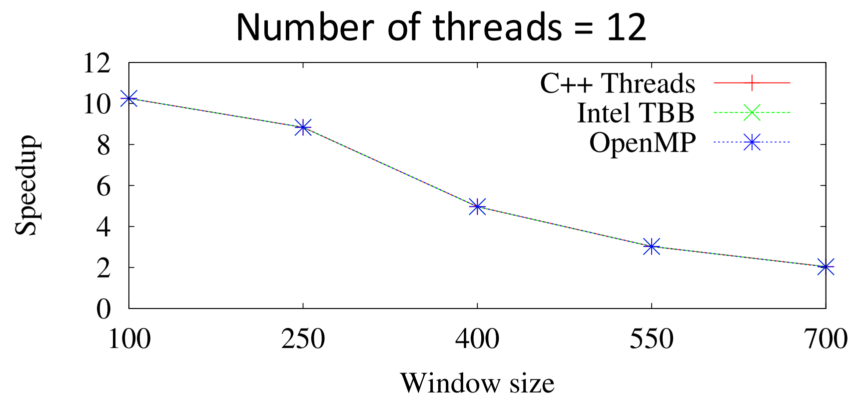
b) The speedup grows with the number of selections -> only selection and evolution functions are parallelized!

Experimental evaluation

- **Windowed-Farm** pattern evaluation using a benchmark computing average window values from an input sensor readings:
  - Sensor sampling frequency is set to 1 kHz
  - Fixed overlapping factor among windows is 90%



(a) Speedup vs. number of threads.



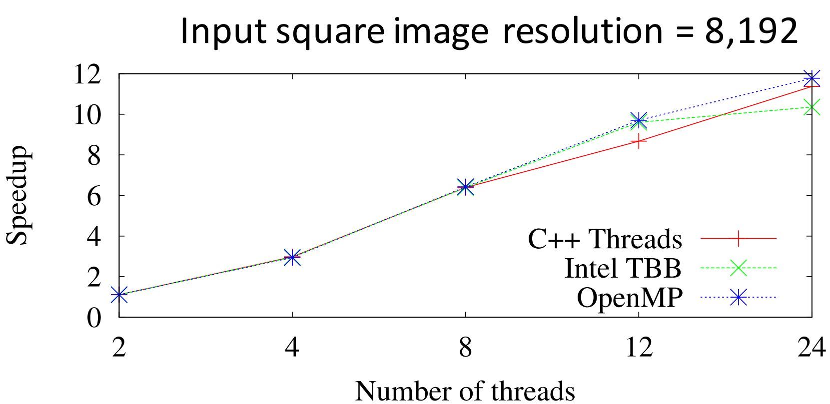
(b) Speedup vs. window size.

a) Good speedup scaling w.r.t number of threads, no difference among frameworks.

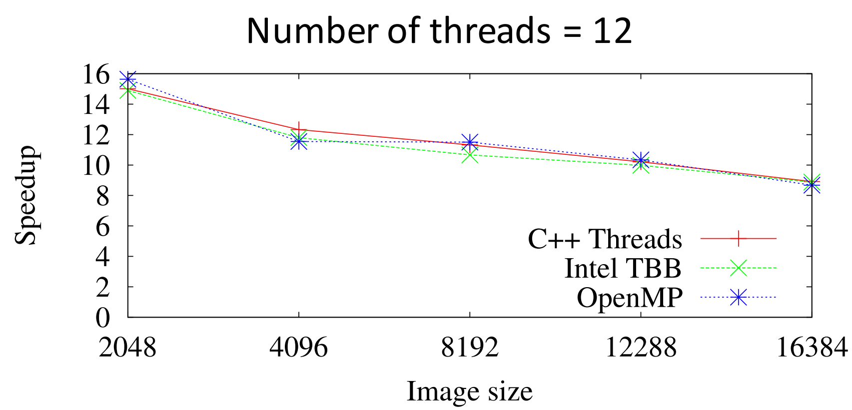
b) The speedup descreases with increasing the window sizes -> number of non-overlapping items grows.

Experimental evaluation

- **Stream-Iteration** pattern evaluation using a benchmark that halves the resolution of a stream of images and delivers them in concrete resolutions.
  - Input square images of resolution 8,192 pixels
  - Output square image resolutions of 128, 512 and 1,024 pixels



(a) Speedup vs. number of threads.



(b) Speedup vs. image size.

a) Good speedup scaling until 12 threads (75% of efficiency), but strong degradation for 24 threads
   → This benchmark is memory-bound when the threads access simultaneously to the input images.

b) Speedup descrease with increasing image sizes -> images accessed by the threads do not completely fit into L2/L3 (30 MB) of the target platform.

Conclusions

- Most programming models are too low-level
  - Ease the parallelization task!
- Patterns hide away the complexity of parallel programming
  - GrPPI is an usable, simple, generic and highlevel parallel pattern interface
  - The overheads of GrPPI are negligible with respect to using directly parallel programming frameworks
  - Advanced patterns can aid in developing complex applications from different specific domains
  - Parallelizing code with GrPPI only requires to modify ~30% the number of lines of the sequential code
- Future work
  - Support for other parallel programming frameworks: FastFlow

THANK YOU!