Project no. 644235

REPHRASE

Research & Innovation Action (RIA)
REFACTORING PARALLEL HETEROGENEOUS RESOURCE-AWARE APPLICATIONS – A SOFTWARE ENGINEERING APPROACH

Software for implementations of advanced patterns
D2.8

Due date of deliverable: December 31st, 2017 (M33)

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

Responsible institution: UNIPI
Editors: Marco Danelutto, UNIPI
Gabriele Mencagli, UNIPI
Massimo Torquati, UNIPI
Tiziano De Matteis, UNIPI

Version 0.2 - December 31st, 2017

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	25/11/17	Manuel F. Dolz	UC3M	GrPPI advanced patterns added
2	11/12/17	G. Mencagli	UNIPI	Overall re-structuring
3	15/12/17	F. Tordini	UNITO	FastFlow GrPPI porting added
4	18/12/17	T. De Matteis	UNIPI	High level FastFlow patterns added
5	21/12/17	G. Mencagli	UNIPI	Windowed high level patterns in FastFlow added
6	24/12/17	F. Tordini	UNITO	Experiments FastFlow/GrPPI added
7	29/12/17	M. Danelutto	UNIPI	Completed initial part
8	31/12/17	Manuel F. Dolz	UC3M	Minor fixes

Executive Summary

This document is the eighth deliverable from WP2 “Pattern-Based Parallel Software Engineering” and it basically describes the features and contents of the software packages building the “Software for implementations of advanced patterns” as described in the amended DoW. In particular, D2.8 integrates the results of the a crucial phase of WP2 (T2.2 “Pattern implementation”) where, according to the amended DoW *we will extend the pattern implementations to cover the advanced patterns that were identified in the second phase of T2.1*, also supporting threading mechanisms (e.g. pthreads, C++11/14 standard), general parallel programming models either as a library (e.g. Intel TBB, FastFlow) or compiler supported (e.g. OpenMP), and GPU programming models (e.g. OpenCL,CUDA).
The deliverable details three different software contributions:

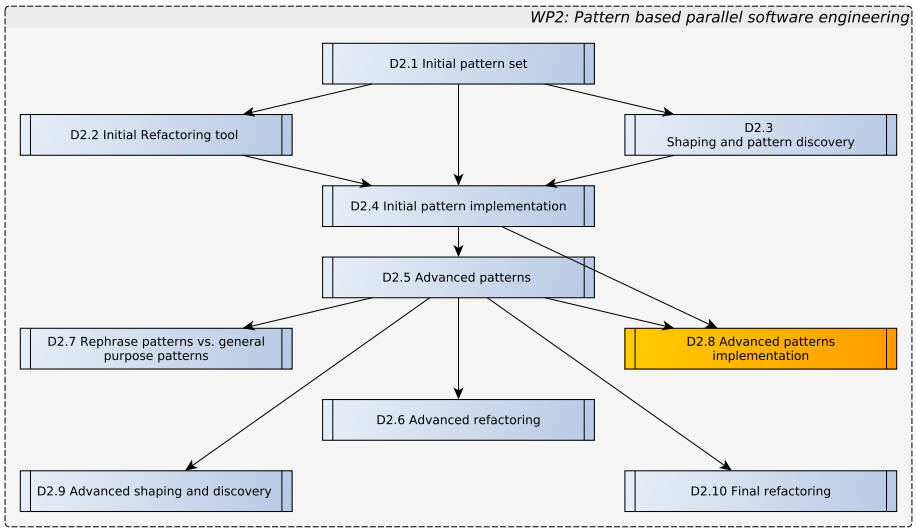
i) a uniform C++11 pattern interface along with an implementation of the advanced patterns in D2.5 on top of C++ threads, OpenMP, Intel TBB,

ii) the implementation of the C++ pattern interface API on top of FastFlow, outlining the pros and cons of the API implementation in terms of performance, and

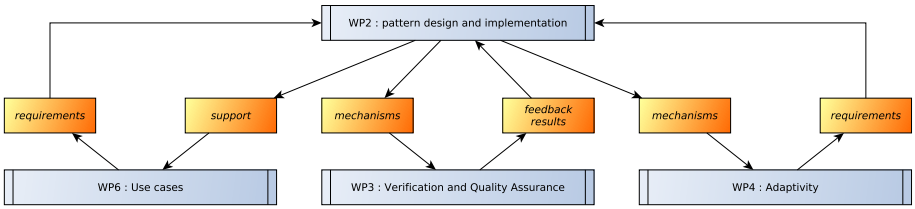
iii) a complete implementation of the D2.5 patterns in FastFlow, provided through standard FastFlow native interface.

The main responsibilities relative to the contributions to this deliverable may be summarized as follows: UNIPI → FastFlow high level pattern, UC3M → GRPPI, UNITO → GRPPI FastFlow interface.

The placement of D2.8 in the WP2 overall deliverable list is summarized by the following schema:



while the strict influences between pattern design and implementation in WP2 and activities in the other major technical workpackages are summarized by the following schema:



Contents

Executive Summary 2

1 Introduction **6**

2 Advanced Patterns in RePhrase **8**

3 Advanced Patterns in GrPPI **10**
3.1 High-level Patterns Interface 10
3.1.1 Pool 10
3.1.2 Stream-Iterator 11
3.1.3 Windowed-Farm 14
3.2 Evaluation of High-level Patterns 16
3.2.1 Usability analysis 17
3.2.2 Performance analysis of the Pool pattern 18
3.2.3 Performance analysis of the Windowed-Farm pattern . . 18
3.2.4 Performance analysis of the Stream-Iterator pattern . . . 21

4 FastFlow back-end for GrPPI **23**
4.1 FastFlow back-end: design 23
4.2 FastFlow back-end: implementation 26
4.3 FastFlow back-end: evaluation 33

5 Advanced Patterns in FastFlow **38**
5.1 High-level Patterns Interface 38
5.1.1 Pool 38
5.1.2 Stream-Iterator 42
5.1.3 Windowed-Farm 44
5.1.4 Keyed-Farm 47
5.2 Evaluation of High-level Patterns 48
5.2.1 Performance analysis of the Windowed-Farm and the Keyed-Farm patterns 50
5.3 Basic Building Blocks Interface 51
5.3.1 Stream Splitter 51
5.3.2 Stream Merger 52

6 Software availability **55**

7 Conclusions and future work **56**

1. Introduction

One of the main aims of the WP2 from the **RePhrase** project is to provide the application programmers with a comprehensive set of parallel patterns that may be used to implement efficient parallel applications. Compared to sequential programming, designing and implementing parallel applications for operating on modern hardware poses a number of new challenges to developers [5]. Communication overheads, load imbalance, poor data locality, improper data layouts, contention in parallel I/O, deadlocks, starvation or the appearance data races in threaded environments are just examples of these challenges. Besides, maintaining and migrating such applications to other parallel platforms demands considerable efforts. Thus, it becomes clear that programmers require an extra expertise, and endeavor, in order to be able to write efficient parallel applications, apart from the knowledge necessary in the application domain.

With the recent emergence of pattern-based programming frameworks, encapsulating algorithmic aspects using a building blocks approach, this aspect has been relieved when targeting parallel platforms [13]. Basically, parallel patterns offer a way to implement robust, readable and portable solutions while hiding away the complexity behind concurrency mechanisms, e.g., thread management, synchronizations or data sharing. Numerous examples of pattern-based programming frameworks, such as SkePU [11], FastFlow [3] or Intel TBB [14], can be found in the literature. Nevertheless, most of these frameworks are not generic enough nor offer unified pattern interfaces [7]. To tackle these issues, the recent interface GrPPI [10], accommodates a unified layer of generic and reusable parallel patterns on the top of existing execution environments and pattern-based frameworks.

In RePhrase, we started considering a basic set of parallel patterns (see D2.1) and we eventually figured out that something was missing in that pattern set. In particular, we individuated a limited number of advanced patterns that could be of great utility in the development of data intensive applications targeting heterogeneous hardware. These new patters have been described in D2.5. Although they can be expressed as non trivial compositions and customization of the patterns in the base RePhrase pattern set, the provision of these patterns as primitive patterns to be instantiated and used simply providing the proper business code parameters makes the task of the application programmers much easier.

Also, within the RePhrase activities, we figured out that a set of compact,

highly composable and very specific *building block* patterns may be efficiently designed to support those data intensive computations not primitively captured by the REPHRASE pattern set. These additional patterns provide an intermediate level of abstraction, lower level with respect to the one provided by the rest of the REPHRASE patterns, but still encapsulate and hide most of the hard to manage details typical of the implementation of efficient data intensive applications.

In this deliverable, we describe the last (but minor bug fixes and tuning) release of the REPHRASE pattern framework, including the implementation of the new advanced and building block patterns and the complete implementation of all those patterns through the GRPPI interface introduced in D2.4.

The rest of this document is organized as follows:

- Chapter 2 briefly recalls the features of the REPHRASE pattern set.
- Chapter 3 extends the GRPPI parallel pattern interface with the **RePhrase** advanced parallel pattern set defined in D2.5. We conduct an experimental evaluation of the GRPPI interface in order to analyze its usability, in terms of lines of code, and its performance, in comparison to the different parallel execution environments currently supported, on different use cases.
- Chapter 4 describes how the FastFlow framework has been ported and integrated into the GRPPI interface. We discuss general design of the integration, some implementation related details as well as some experiments assessing the whole implementation and evaluating the performance penalties/advantages w.r.t. the usage of (native) FastFlow.
- Chapter 5 discusses the native FastFlow support of the advanced patterns identified in D2.5 as well as of the building block data intensive patterns. Part of the patterns discussed in this part of the deliverable, in particular those eventually exploited within the REPHRASE use cases, will be integrated into GRPPI by the end of the project with the native FastFlow interface. The native FastFlow implementation has been designed with performance and efficiency in mind, rather than targeting expressive power and the most recent C++-style programming model compliance.
- Chapter 6 we eventually detail where the software discussed in the deliverable may be found.

2. Advanced Patterns in RePhrase

One of the main aims of the WP2 in the **RePhrase** project is to provide the application programmers with a comprehensive set of parallel patterns that may be used to implement efficient parallel applications. During the **RePhrase** project, the individual study of the use cases in WP6, and of other case studies of interest to the project's participants, has revealed the need of a set "high-level" avanced patterns to support such data-intensive computations usually featuring complex algorithmic structures. The set of "high-level" patterns was defined in D2.5 and extends the set of basic patterns presented in D2.1. Concretely, we refer to "advanced" patterns those designed for scenarios where the basic patterns do not match any of these constructions or have to be composed in a very complex way that deserves to be provided as a new pattern *per se*. More specifically, the advanced patterns of the project were discovered in various domains like in evolutionary and symbolic computing [12], wireless sensor networks [8], and from the real-time data stream processing landscape [6].

According to the specification provided in D2.5, the set of advanced patterns is composed of two different parts: *i)* the first part consists of the so-called high-level parallel patterns, that can be directly utilized to model complex application tasks that require data-intensive parallelization not easy to implemented using exclusively the patterns in the basic set (the ones in D2.1); *ii)* the second part consists of a set of building block patterns that can be considered escape solutions to be used to implement data-intensive parallel patterns in special cases not covered by the basic pattern compositions or by the higher level advanced patterns. Tab. 2.1 shows the list of the REPHRASE high-level parallel patterns and their implementation with the various back-ends behind the GrPPi interface, along with the implementation using the native FastFlow interface.

As it is evident from the table, each pattern is implemented either in the unified GrPPI interface with at least one (or in some cases more than one) back-end, or it has been implemented in the native FastFlow parallel programming framework. This is fully compliant with the REPHRASE DoW specifications (where GRPPI was not originally included) and with what we stated in the initial WP2 deliverables, where it was explicitly stated that *a)* we were not aiming at providing all patterns in all back-ends and *b)* we were not aiming at providing interoperability between different back-ends, in the perspective of leaving the application (use

Table 2.1: Advanced parallel patterns vs. Frameworks supported through GRPPI.

	GRPPI						Native
	Sequential	**OMP**	**TBB**	**Threads**	**FastFlow**	**CUDA Thrust**	**FastFlow**
Pool	✓	✓	✓	✓	✗	✗	✓
Convolution	✗	✗	✗	✗	✗	✗	✓
Stream-Iterator	✓	✓	✓	✓	✗	✗	✓
Windowed-Farm	✓	✓	✓	✓	✗	✗	✓
Keyed-Farm	✗	✗	✗	✗	✗	✗	✓

case) programmers to pick up the better pattern set/back-end combination.

In addition to high level patterns, REPHRASE advanced pattern set defined in D2.5 includes the basic building block patterns: *inline stream generator*, *stream merger*, *stream splitter*, *stream tupler* and *stream de-tupler*. These patterns, used to build complex stream processing networks are all supported by the FastFlow native implementation.

3. Advanced Patterns in GRPPI

In this chapter, we discuss the extension of the generic and reusable parallel pattern interface GRPPI previously presented in D2.4, with the **RePhrase** advanced parallel patterns. Recall that this interface, implemented in C++, is a parallel pattern library targeting threading-aware mechanisms (e.g. pthreads, C++11/14 standard), general parallel programming models either as a library (e.g. Intel TBB, FastFlow) or compiler supported (e.g. OpenMP), and GPU programming models (e.g. CUDA Thrust). In this stage, GRPPI supports the advanced set of patterns, previously introduced in D2.5, with different programming models as specified Table 2.1. We follow the same classification of patterns listed in deliverable D2.5 for describing the interfaces proposed.

3.1 High-level Patterns Interface

In this section, we describe three advanced **RePhrase** patterns (Pool, Windowed-Farm and Stream-Iterator) that have been incorporated in GRPPI.

3.1.1 Pool

This pattern models the evolution of a population of individuals matching many evolutionary computing algorithms in the state-of-the-art [1]. Specifically, the Pool pattern is comprised of four different functions that are applied iteratively to a population P of individuals of type α. First, the *selection* function S: $\alpha^* \rightarrow \alpha^*$ selects a subset of individuals belonging to P. Next, the selected individuals are processed by means of the *evolution* function E: $\alpha^* \rightarrow \alpha^*$, which may produce any number of new or modified individuals. The resulting set of individuals computed by E are filtered through a *filter* function F: $\alpha^* \rightarrow \alpha^*$, and eventually inserted into the population. Finally, the *termination* function T: $\alpha^* \rightarrow \{true, false\}$ determines in each iteration whether the evolution process should be finished or continued. To guarantee the correctness of the parallel version of this pattern, both functions S and E should be pure, i.e., they can be computed in parallel with no side effects. More details about this pattern can be found in Deliverable 2.5, Section 2.1.

Interface

The GrPPI interface designed for the Pool pattern, shown in Listing 3.1, receives the execution model, the population (`popul`), the selection (`select`), evolving (`evolve`), filtering (`filter`) and termination (`term`) functions, and the number of selections that will be performed. Initially, the parallel pattern implementation of GrPPI divides the number of selections among the concurrent processing entities that will select and evolve the population individuals. Afterwards, the resulting individuals are merged and forwarded to the sequential filtering and termination functions. Finally, only if the termination condition is met, the Pool parallel pattern finishes and delivers the resulting population. On the contrary, the whole process is repeated again with the evolved population.

The parallelism of this pattern is controlled via the execution model parameter, which can be set to operate in sequential or in parallel, through the different supported frameworks; e.g. to use C++ threads, the parameter should be set to `parallel_execution_native`. In this case, any execution model can optionally receive, as an argument, the number of entities to be used for the parallel execution, e.g., `parallel_execution_native{6}` would use 6 worker threads. If this argument is not given, the interface takes by default the number of threads set by the underlying platform.

Listing 3.1: 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);
```

Example

As an example, Listing 3.2 shows an instance of a Pool used for implementing the *Travelling Salesmann Problem*. As can be seen, the selection lambda function receives as parameter the population and selects an individual randomly. Next, the evolution function performs a series of swaps among the individual's representation. In the filtering lambda function, the evolved individual with the best score is selected. Finally, the termination function determines whether an individual has an score lower than a given threshold in order to finalize the execution. The last parameter, 50, indicates the number of maximum selections that the Pool pattern must perform in each iteration.

3.1.2 Stream-Iterator

This pattern computes in parallel a filter over an input stream of type α stream, that is passes to the output stream of type α stream only those input data items passed by a given boolean "filter" function (predicate) $\mathcal{P} : \alpha \rightarrow \{true, false\}$. During the refactoring process, the introduction of this pattern may requires the

Listing 3.2: Usage example of the Pool pattern.

```
pool(ex, population,
  // Selection
  [](std::vector<std::vector<int>> population){
    auto sel = rand() % population.size();
    return make_pair(sel, population[sel]);
  },
  // Evolution
  [](std::vector<int> individual){
    for(int i= 0; i<num_swaps;i++){
      int swap = rand() % (individual.size()-1);
      auto aux = individual[swap];
      individual[swap] = individual[swap+1];
      individual[swap+1] = aux;
      auto k = aux;
    }
    return individual;
  },
  // Filtering
  [&](std::vector<std::vector<int>> population,
      std::vector<std::vector<int>> & evolved){
    int bestEv = 0;
    int costEv=1000;
    for(int i = 0; i < evolved.size(); i++){
      int currentcost = computeCost(evolved[i]);
      if(currentcost < costEv) { bestEv = i; costEv = currentcost;}
    }
    return evolved[bestEv];
  },
  // Termination
  [&](std::vector<int> individual){
    auto cost = computeCost(individual);
    if(cost <= threshold)) return true;
    return false;
  },
  // Number of selections
  50
);
```

introduction of a stream generator and collapser patterns. More details about this pattern can be found in Deliverable 2.1, Section 3.2.3.

Interface

The GRPPI interface for the Stream-Iterator pattern, detailed in Listing 3.3, takes the execution model, the stream consumer (`in`), the kernel (`task`) and the producer (`out`) functions. This pattern also receives two boolean functions: the termination (`term`) and output guard (`guard`) functions. In the first step, the `in` function reads items from the input stream and a worker thread executes the kernel `task` function for each item. Next, the termination function `term` is evaluated with the resulting item to determine if the kernel should be re-executed on the same input item. Additionally, the output `guard` function decides whether an item should be delivered to the output stream or not.

Listing 3.3: Stream-Iterator interface.

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

As stated, the parallelism of the Stream-Iterator pattern is only obtained when it is composed with a basic GRPPI parallel pattern, e.g., Farm or Pipeline. As an example of composition, the code in Listing 3.4 implements a Stream-Iterator, in which the kernel `task` function has been composed with the Pipeline pattern. Therefore, the kernel is computed in parallel by 2 worker threads. Note that the `optional`, as for the return type in the consumer function lambda, is used to indicate the end of the stream when constructed without arguments. As can be seen, thanks to GRPPI, it is possible to compose advanced with basic parallel patterns in order to increase the parallelism degree.

Listing 3.4: Example of Stream-Iterator-Pipeline composition.

```
iteration_multi_out( parallel_execution_native{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; }
    ),
    [&]( int e ){ os << e << endl; }, // Producer function
    [] ( int e ){ return e < 100; }, // Termination function
    [] ( int e ){ return e % 2 == 0; } // Output guard function
);
```

Example

Listing 3.5 shows an example of a Stream-Iterator implements an application that is intended to reduce the resolution of the images appearing in the input stream, and to produce images with concrete resolutions to the output stream. As observed in the code, the first parameter of the Stream-Iterator pattern corresponds with the execution model; in this case the `parallel_execution_native`, which corresponds with the C++ Threads back end. The next parameter is the consumer function, which reads images from a given input stream. Afterwards the kernel function, computed by means of a Farm pattern, reduces in parallel the resolution of the input images to the half. The producer function of the pattern writes the resulting images the output stream. The last two functions in the Stream-Iterator pattern correspond to the termination and output guard. Basically the termination determines whether the iterations on a same image should be finished (when the image size is lower than 128×128 pixels) or not. The output guard dictates if the image should be generated to the output stream during an intermediate iteration, i.e., when the image has a concrete resolution as defined in its condition.

3.1.3 Windowed-Farm

This stream-oriented pattern delivers "windows" of processed items to the output stream. Basically, this pattern applies the function WF over consecutive contiguous collections of x input items of type α and delivers the resulting windows of y items of type β to the output stream. Optionally, these windows can have an slide, i.e., the number of items in the window w_i that are also part of the window w_{i+1}. The parallelization of this pattern requires a pure function WF: $\alpha^* \rightarrow \beta^*$ for processing item collections. More details about this pattern can be found in Deliverable 2.1, Section 2.4.1.

Interface

The interface for the Windowed-Farm pattern, described in Listing 3.6, receives the execution model, the stream consumer (`in`), the Farm (`task`) and the producer (`out`) functions. This pattern also receives the size and the window slide.[1] Specifically, the `in` function reads from the input stream as many items as required to fill the window buffer. Next, this buffer is forwarded to one of the concurrent entities, which will compute the function `task` in a Farm-like fashion. Therefore, the parallel implementation of this GRPPI pattern is offered by the Farm construction. Finally, the items collections resulting from the `task` function are delivered to the output stream. Note that, depending on the user requirements, this pattern can deliver items windows in an ordered way by properly configuring the execution model.

[1] Note that while the current Windowed-Farm pattern only supports count-based windows, in the future we plan to extend its interface to cover time-based, slide-by-tuple and delta-based windowing models.

Listing 3.5: Usage example of the Stream-Iterator.

```
iteration_multi_out( parallel_execution_native{},
  // Consumer
  [&]() -> optional<std::vector<std::vector<short>>>{
    a--;
    auto input = read_image();
    if ( a == 0 )
      return {};
    else
      return {input};
  },
  // Kernel
  farm( 8, [](auto image) {
    std::vector<std::vector<short>> out((image.size()/2),
                std::vector<short>(image[0].size()/2) );
    int currsize = image.size();
    for(int i =0, ni = 0 ; i< currsize; i+=2, ni++)
      for(int j =0, nj = 0 ; j< currsize; j+=2, nj++)
        out[ni][nj] = (image[i][j]   + image[i+1][j] +
                       image[i][j+1] + image[i+1][j+1])/4;
    return out;
  }),
  // Producer
  [&](std::vector<std::vector<short>> image){
    out.write(image);
  },
  // Termination
  [](std::vector<std::vector<short>> image){
    return image.size() > 128 ? true : false;
  },
  // Output guard
  [](std::vector<std::vector<short>> image){
    return (image.size() == 1024 || image.size() == 512 ||
            image.size() == 128) ? true : false;
  }
);
```

Listing 3.7: Usage example of the Windowed-Farm pattern.

```
window_farm( parallel_execution_native{8},
  // Consumer
  [&]() -> optional<int> {
    n++;
    if (n != 100000)
      return {sensor.read()};
    else
      return {};
  },
  // Farm kernel task
  [](auto window){
    int total = 0;
    for(auto it = window.begin(); it < window.end(); it++)
      total += (*it);
    return total/window.size();
  },
  // Producer
  [&](int res) {
    std::cout << res << std::endl;
  },
  // Window size
  100,
  // Slide
  10
);
```

Listing 3.6: Windowed-Farm interface.

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

Example

Listing 3.7 presents an example of a Windowed-Farm pattern used to compute average window values from an emulated sensor readings. As can be seen in the code, the first parameter corresponds to the C++ Threads GRPPI back end executed with 8 worker threads. Next, the kernel task computes in parallel the average of the window values, while the consumer is in charge of printing out these results. The las two parameters determine the window size and the slide. Thanks to this pattern, the implementation of this use case can be largely simplified.

All in all, these domain-specific algorithms can be implemented using the proposed GRPPI interfaces for the advanced patterns with relatively small efforts.

3.2 Evaluation of High-level Patterns

In this section, we perform an experimental evaluation of the three novel advanced patterns from the usability and performance points of view. To do so, we use the

following hardware and software components:

- *Target platform.* The evaluation has been carried out on a server platform comprised of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM. The OS is a Linux Ubuntu 14.04.2 LTS with the kernel 3.13.0-57.

- *Software.* To develop the parallel versions and to implement the proposed interfaces, we leveraged the execution environments C++11 threads and OpenMP 4.5, and the pattern-based parallel framework Intel Threading Building Blocks (TBB). The C++ compiler used to assemble GRPPI is GCC v5.0.

- *Use cases.* To evaluate the advanced patterns, we use three different synthetic use cases targeting problems from different domains.
  - The Pool pattern has been evaluated on a benchmark that solves the *traveling salesman* problem (TSP) using a regular evolutionary algorithm. This NP-problem computes the shortest possible route among different cities, visiting them only once and returning to the origin city.
  - To evaluate the Windowed-Farm, we use a benchmark that computes average window values from an emulated sensor readings.
  - For the Stream-Iterator, we leverage a benchmark that reduces the resolution of the images appearing in the input stream, and produces the images with concrete resolutions to the output stream.

In the following sections, we analyze the usability, in terms of lines of code, and the performance of the GRPPI advanced patterns using the above-mentioned benchmarks with varying configurations of parallelism degree, problem size and execution frameworks.

3.2.1 Usability analysis

In this section we analyze the usability and flexibility of the advanced pattern interfaces. To analyze these aspects, we assess the number of modified lines of code (LOCs) required to implement the parallel versions of the use case algorithms. Then, we compare the modified LOCs leveraging the GRPPI interface with respect to using directly the supported frameworks. Table 3.1 summarizes the percentage of modified LOCs in the sequential algorithm in order to implement the parallel versions of the use cases algorithms. As observed, the OpenMP and TBB versions require less LOCs, given that these frameworks provide high-level interfaces hiding away the complexity behind concurrency mechanisms. For instance, OpenMP 4.5 offers the `depend` clause in `task` directives which enforces additional constraints on the scheduling of tasks. However, the analogous implementation in C++ threads requires the use of explicit communication channels (e.g. multiple-produce/multiple-consumer queues) and synchronization mechanisms (e.g. locks, condition variables and atomic variables). On the other hand, using the GRPPI

interface for parallelizing a given application is simpler than using directly the above-mentioned programming frameworks. On average, the LOCs that have to be modified in order to incorporate an advanced GRPPI pattern, is 28 %. An additional advantage of GRPPI is its capability to easily switch among execution frameworks, since it is only required to replace a single argument in the pattern function call.

Table 3.1: Percentage of modified lines of code 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 %

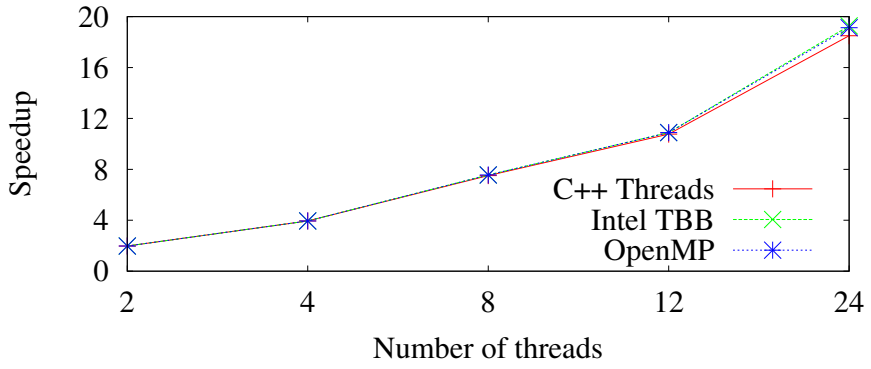
3.2.2 Performance analysis of the Pool pattern

Next, we evaluate the Pool pattern on a benchmark that solves the TSP problem using a population of 50 individuals representing feasible routes. We also set the benchmark to perform a total of 200 iterations, each of them making 200 selections. Figure 3.1a shows the performance gains when varying the number of threads, from 2 to 24, and using the three available GRPPI back-ends: C++ threads, OpenMP and Intel TBB, with respect to the sequential version. As can be seen, the speedup increases roughly at a linear rate when increasing the number of threads for all frameworks. Concretely, we observe that between 2 and 12 threads the efficiency is sustained in the range of 91 %–98 %. However, for 24 threads the frameworks OpenMP and Intel TBB deliver an efficiency of 80 %, while for C++ threads it slightly decreases to 77 %. This is mainly due to the better resource usage made by the OpenMP and Intel TBB runtime schedulers.

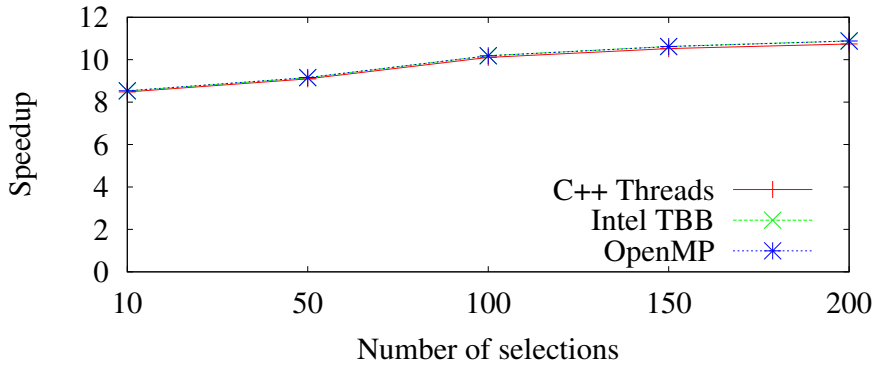
As a complementary evaluation, we set the number of threads to 12 and vary the number of selections from 10 and 200. According to the results shown in Figure 3.1b, the speedup grows hand in hand with the number of selections, since the Pool pattern only parallelizes the selection and evolution functions. This indicates that increasing the number of selections improves the load balance among the worker threads and pays off the parallelization overheads related to thread synchronizations and communications.

3.2.3 Performance analysis of the Windowed-Farm pattern

In this section, we evaluate the performance of the Windowed-Farm using a synthetic benchmark that computes average window values from an input stream of sensor readings. Specifically, the sensor in this benchmark has been configured to read samples at 1 kHz and the pattern window size has been set to 100 elements with 90 % of overlap among windows. Figure 3.2a shows the speedup when the

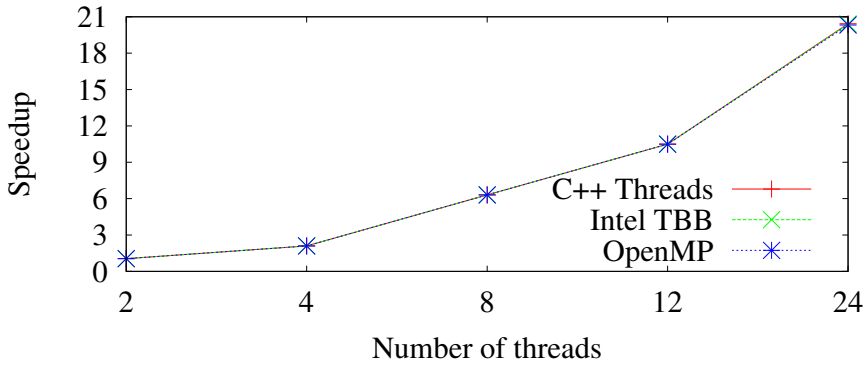


(a) Speedup vs. number of threads.

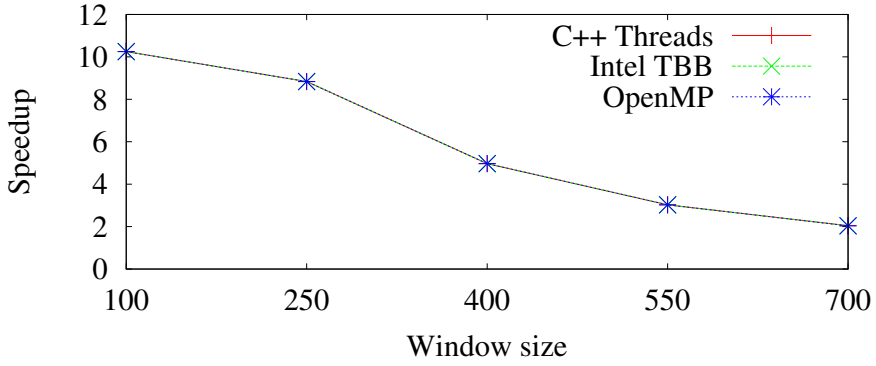


(b) Speedup vs. number of selections.

Figure 3.1: Pool speedup varying with varying number of threads and selections.



(a) Speedup vs. number of threads.



(b) Speedup vs. window size.

Figure 3.2: Windowed-Farm speedup with varying number of threads and window size.

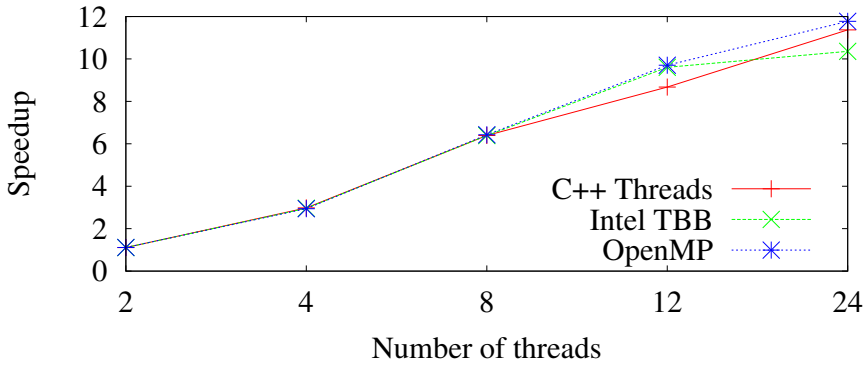
number of threads increases from 2 to 24. The main observation is that all execution frameworks scale with the increasing number of threads and behave similarly, given that the OpenMP and Intel TBB runtime schedulers do not provide any major advantage over the C++ threads implementation in this concrete use case. This is because the internal Farm pattern leads, by nature, to well balanced workloads among threads. Note that a Farm is comprised of a pool of threads that constantly retrieve items from the input stream and apply the same function over them. On the other hand, we also observe an almost linear scaling for increasing number of threads. This is mainly caused because the Farm pattern can theoretically scale up to $\frac{T_f}{T_a}$, being T_f the computation time of the window average value and T_a the interarrival time of windows in the input stream. To demonstrate this strong scaling, we experimentally measured the computation time of the average function, which was, on average, 220 ms and the interarrival window time that was 10 ms. Therefore, applying the aforementioned formula, we get 22 as for the maximum theoretical speedup.

As an additional experiment, we evaluate the behavior of the Windowed-Farm pattern when increasing the window size, using 12 threads and the aforementioned configuration that uses a fixed overlapping factor of 90 %. As can be observed from Figure 3.2b, the speedup decreases for increasing window sizes, as the number of non-overlapping items among windows also increases. This basically occurs because the interarrival time of window T_a increases, restricting proportionally the maximum parallelism degree.

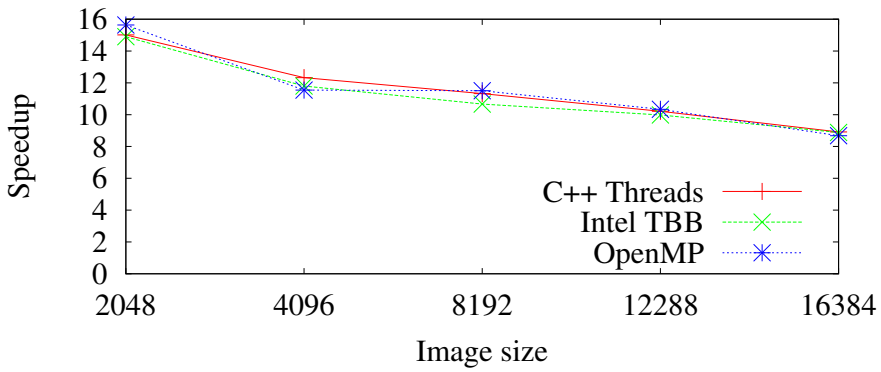
3.2.4 Performance analysis of the Stream-Iterator pattern

Finally, we analyze the performance of the GRPPI Stream-Iterator pattern using the above-mentioned benchmark, in charge of processing square images and halving their sizes on each iteration until reaching concrete resolutions. Specifically, the size of the input images is fixed to 8,192 pixels, and the output images, for each input, have sizes of 128, 512 and 1,024. Figure 3.3a illustrates the benchmark speedup when varying the number of threads from 2 to 24 for the different GRPPI back-ends. In this case, when the number of threads ranges between 2 and 12, the efficiency attained is roughly 75 %, while for 24 this is degraded to 48 % for all programming frameworks. This effect is mainly caused by the fact that each of the threads involved in the Farm pattern, part of the Stream-Iterator, are simultaneously accessing to different input images. Therefore, these memory accesses become a bottleneck due to constant cache misses when the threads perform the computation of the `task` function of the pattern. In general, these results suggest a memory bandwidth limitation in this particular benchmark.

To gain insights into the performance degradation detected in the previous analysis, we perform an additional experiment in which we set the number of threads to 24 and vary the input image sizes from 2,048 to 16,384. Figure 3.3b depicts the performance gains for the different execution frameworks when varying the image size in the preceding range. Again, we observe a slight speedup decrease for increasing image sizes, which confirms our prior impressions. As an example, if we assume 22 worker threads in the internal Farm pattern, individually processing images with resolution of 2,048×2,048 pixels (represented with matrices of integers), these require about 352 MiB of memory. Therefore, not fitting in any of the available cache levels and leading to an increased L2/L3 cache miss rate when they are simultaneously accessed. All in all, this issue is mainly due to the inherent memory-bound nature of this specific use case.



(a) Speedup vs. number of threads.



(b) Speedup vs. image size.

Figure 3.3: Stream-Iterator speedup with varying number of threads and image size.

4. FastFlow back-end for GRPPI

In this section we discuss how the FastFlow framework has been ported and integrated into the GRPPI interface.

FastFlow natively supports high-level parallel programming patterns for dealing with stream parallelism and data parallelism, built on top of a *lock-free* run-time designed to boost performance on multi-core and many-cores architectures.

This makes it perfectly suitable for being used as a back-end in the GRPPI interface. FastFlow provides different high-level patterns as well as different lover level patterns (parallel applications building blocks) that may be both used to implement quite efficient parallel applications. While simple usage of high level patterns in FastFlow does not require sensibly different expertise than the one required using the same patterns through GRPPI, the FastFlow API does not reach GRPPI's level of abstraction. In this section, we will go through the process of implementing GRPPI's interface by means of FastFlow patterns, with an evaluation of the performance of the porting of FastFlow to GRPPI, comparing the pure FastFlow implementation against the wrapped one, in order to identify overheads and possible performance differences.

4.1 FastFlow back-end: design

GRPPI provides parallel patterns to support both stream processing and data-intensive applications. These parallel patterns can be nested and composed together, in order to model more complex behaviors. As it was presented in deliverable D2.4 (Sec. 3), GRPPI currently supports the following base patterns:

Listing 4.1: GRPPI's Pipeline interface

```
template <typename Execution, typename Generator,
typename ... Transformers, requires_execution_supported<Execution> = 0>
void pipeline(const Execution & ex, Generator && generate_op,
        Transformers && ... transform_ops) {
    ex.pipeline(std::forward<Generator>(generate_op),
        std::forward<Transformers>(transform_ops)...);
}
```

- *Data-parallel* patterns
  - Map
  - Reduce
  - Map/Reduce
  - Stencil
- *Task-parallel* patterns
  - Divide-and-Conquer
- *Stream-parallel* patterns
  - Pipeline
  - Farm
  - Stream filter
  - Stream reduction
  - Stream iteration

The parallel semantics associated to these patterns will not be further discussed here, because it was already explained in deliverable D2.1.

FastFlow natively supports most of the aforementioned skeletons, which are implemented by properly combining instances of the `ff_node` object, which is the main building block upon which every pattern can be designed. In particular, the `ff_node` object implements the abstraction of an independent concurrent activity receiving input tasks from an input queue, processing them and delivering the related results to the an output queue. For instance, `ff_pipeline` and `ff_farm` are FastFlow's core patterns built on top of `ff_node`, which represent the most basic and generic stream parallel patterns and *de facto* orchestrate different kind of concurrent activities: stages, in the case of the pipeline pattern, or scheduler, workers and result collector, in the case of the farm pattern.

In its most recent fully C++11 compliant API FastFlow introduced enhanced features that support a better control of the typing system of programming patterns. Starting from its building blocks, the `ff_node_t<TypeIn, TypeOut>` becomes a typed abstract container for parallel activities, useful to enforce better type checking on streaming patterns, while among the high-level patterns `ff_Farm<TypeIn, TypeOut>` and `ff_Pipe<TypeIn, TypeOut>` provide typed Farms and Pipelines.

GRPPI takes full advantage of modern C++ features, making an extensive use of template meta-programming and generic programming concepts, that surely help code minimization and leverage automatic compile-time optimizations. The transformations applied to input data are described using C++ lambda functions, where the callable objects are forwarded via template function parameters, and the body of the lambda will be completely in-lined inside a specific specialization of the template function that accepts it. Here the compiler will optimize the whole body of the function: in performance-critical applications, this solution certainly brings remarkable benefits.

Pattern interfaces are described as function templates, making them more flexible and usable with any (input, output) data type. The extensive use of variadic templates allows an arbitrary number of data structures to be used by a pattern, and also facilitates handling an arbitrary number of stages/components as typical in a Pipeline pattern, by receiving a collection of callable objects passed as arguments to the function: template overloading guarantees that the right case is matched for

Listing 4.2: Parallel execution with FastFlow

```
class parallel_execution_ff {
public:
    parallel_execution_ff{get_physical_cores()}
    { }
    parallel_execution_ff(int degree) :
        concurrency_degree_{degree}
    { }

    // setters and getters...

    template <typename InputIterator, typename Identity,
    typename Combiner>
    auto reduce(InputIterator first,
            std::size_t sequence_size,
            Identity&& identity,
            Combiner&& combine_op) const;

    // other patterns...

    template <typename Generator, typename ... Transformers>
    void pipeline(Generator&& generate_op,
            Transformers&& ... transform_op) const;
}
```

every combination of Pipeline stages.

Each pattern declaration contains an `Execution` type, which represents the back-end that will eventually power pattern execution (see listing 4.1). For instance, it can be set to operate sequentially, or in parallel by means of one of the supported parallel frameworks. The FastFlow back-end is fully compliant with these techniques: it has been injected among the supported execution type, and its declaration updated among the type traits and meta-functions that GRPPI applies to assert correctness of execution.

The programming framework to be used to support the execution of an high level GRPPI pattern is described specifying the proper `parallel_execution_*` object as a pattern parameter. Each `parallel_execution_*` object provides methods for interacting with the underlying parallel framework and contains the actual declaration of patterns.

Listing 4.2 reports an extract of the `parallel_execution_ff` object, which reflects the way the other back-ends have actually been designed: by default the concurrency degree is set to the number of available physical cores, but it could be changed via proper setters and getters methods or using the specific constructor getting the concurrency degree as a parameter.

Stream parallel and data parallel patterns are declared and implemented within the execution object: linking FastFlow to GRPPI's data parallel patterns is a rather straightforward job, as they can all be built on top FastFlow's `ParallelFor` skeleton. Stream parallel patterns require a slightly bigger effort, as the existing Pipeline pattern has to be used to support the "pattern-matching" mechanism that

Listing 4.3: Map pattern for FastFlow back-end

```
template <typename ... InputIterators, typename OutputIterator,
typename Transformer>
void parallel_execution_ff::map(std::tuple<InputIterators...> firsts,
    OutputIterator first_out,
    std::size_t sequence_size,
    Transformer transform_op) const {
    ff::ParallelFor pf(concurrency_degree_, true);

    pf.parallel_for(0, sequence_size,
        [&](const long it_) {
            *(first_out+it_) =
                apply_iterators_indexed(transform_op, firsts, it_);
    }, concurrency_degree_);
}
```

drives template overloading, leading to the definition of the streaming patterns as composition of Pipelines with a minimum of 3 stages.

Concerning the Divide-and-Conquer (DAC) pattern, FastFlow' DAC pattern does not fully correspond to GRPPI's one, thus the porting requires some adjustments, due to differences in their programming interface. Further details will be given in the following section.

4.2 FastFlow back-end: implementation

Data Parallel Patterns

FastFlow provides the `ParallelFor` and `ParallelForReduce` skeletons, designed to efficiently exploit loop parallelism over input data collections. The FastFlow `ParallelFor` has obtained comparable, or even better performance results with respect to those achieved with well-known frameworks, such as OpenMP or Intel TBB [2, 4], which were already part of GRPPI's back-end. As previously mentioned, a FastFlow back-end to data parallel patterns is rather straightforward, as they can all be built on top of FastFlow's `ParallelFor`, even though it requires some caution on pointers arithmetic.

Listing 4.3 shows the Map pattern built on top of the `ParallelFor`: the object is constructed with the proper concurrency degree, enabling a non-blocking run-time. The grain size also enables a dynamic scheduling of tasks to the threads, and chunks of no more than *grain* iterations at a time are computed by each thread, while the chunk of data is assigned to worker threads dynamically.

The `ParallelFor` body traverses each chunk with indexes ranges, which are controlled using pointer arithmetic: the `apply_ iterators_indexed` is a meta-function provided by GRPPI that applies a callable object (`transform_op`) to the values obtained from the iterators in a tuple by indexing. Results of the callable are written directly into the output data structure.

Listing 4.4: Map/Reduce pattern for FastFlow back-end

```
template <typename ... InputIterators, typename Identity,
typename Transformer, typename Combiner>
auto parallel_execution_ff::map_reduce(
    std::tuple<InputIterators...> firsts, std::size_t sequence_size,
    Identity && identity, Transformer && transform_op,
    Combiner && combine_op) const {
    ff::ParallelForReduce<Identity> pfr(concurrency_degree_, true);
    Identity vaR = identity, var_id = identity;
    std::vector<Identity> partials_(sequence_size);

    // Map function
    auto Map = [&](const long it_) {
            partials_[it_] = apply_iterators_indexed(transform_op, firsts,
               (cont.) it_);
    };
    // Reduce function - executed in parallel
    auto Reduce = [&](const long it_, Identity &vaR) {
        vaR = combine_op( vaR, partials_[it_] );
    };
    // Final reduce - executed sequentially
    auto final_red = [&](Identity a, Identity b) {
        vaR = combine_op(a, b);
    };

    pfr.parallel_for(0, sequence_size, Map, concurrency_degree_ );
    pfr.parallel_reduce(vaR, var_id, 0, sequence_size,
                            Reduce, final_red, concurrency_degree_);

    return vaR;
}
```

Similarly, the Reduce pattern is built on top of the `ParallelFor Reduce`: the reduction is executed in two phases: a partial reduction phase first runs in parallel, while the second phase reduces partial results in series, returning the final output. Iterations are scheduled to worker threads in blocks of at least *grain* iterations. Listing 4.4 shows the Reduce pattern FastFlow implementation for GRPPI.

The Map/Reduce pattern computes a Map-like operation on the input data set, and then applies a Reduce operation on intermediate results. It exploits the intrinsic parallelism provided by the Map and the Reduce patterns, and the implementation is thus a combination of two phases: the Map phase stores its result in a temporary collections, which then undergoes the Reduce phase, and the final result is returned (see listing 4.4). It is worth pointing out that this pattern is not the titled Google's MapReduce, which exploits *key-value* pairs to compute problems that can be parallelized by mapping a function over a given data set or stream of data, and then combining the results.

The Stencil pattern (see Listing 4.5) is a generalization of the Map pattern, except that it also performs transformations on a set of neighbors in a given coordinate of the input data set. Just like the Map pattern, it is implemented on top of the `ParallelFor`, with the difference that for each element, the result of the

Listing 4.5: Stencil pattern for FastFlow back-end

```
template <typename ... InputIterators, typename OutputIterator,
typename StencilTransformer, typename Neighbourhood>
void parallel_execution_ff::stencil(std::tuple<InputIterators...> firsts,
      OutputIterator first_out,
      std::size_t sequence_size,
      StencilTransformer&& transform_op,
      Neighbourhood&& neighbour_op) const {
      ff::ParallelFor pf(concurrency_degree_, true);

      pf.parallel_for(0, sequence_size,
      [&](const long it_) {
                        const auto first_it = std::get<0>(firsts);
                        auto next_chunks = iterators_next(firsts, it_);
                        *(first_out+it_) = transform_op( (first_it+it_),
                              apply_increment(neighbour_op, next_chunks)
                        );
      }, concurrency_degree_);
}
```

transformation is combined with the result of a function applied to the neighboring elements of the current one. The pattern can deal with multiple input data sets, which can be specified by mean of variadic parameter in its declaration.

Stream Parallel Patterns

In streaming parallelism, the *end-of-stream* is a special event used to manage termination that has to be orderly notified to the involved concurrent activities, so that they know the computation is terminated and they could start the appropriate closing actions. For this reason, a tag or a marker that identifies the end of data stream is required. GRPPI uses the `optional<T>` data type, officially introduced in C++17 but already available in C++14 under the *experimental* namespace. This data type is basically a class template that manages an optional contained value, i.e. a value that may or may not be present. The empty `optional` is used to represent the EOS.

On the other hand, FastFlow propagates special tags through its actors to notify special events, such as the termination of the data stream, or items to be discarded. For instance, the end-of-stream is identified with the tag `(OutType *) EOS`. This incongruity between the two interfaces has been handled at the lower level, by wrapping the `ff_node_t` object at the base of patterns' implementation.

By exploiting template overloading, an `FFNode` has been declared for three basic situations in a Pipeline pattern: listing 4.6 reports the wrappers for the `ff_node_t` as a template classes, where input and output data types are specified, as well as the actual callable objects. The *first* stage traces a stream generator, it does not need any input data type to be specified and sends items into the data stream as long as its callable returns a value.

The end-of-stream is handled through the `check` variable, roughly in the same

Listing 4.6: FFNode for streaming patterns (initial)

```
template <typename TSout, typename F>
struct FFNode<void,TSout,L> : ff_node {
    L callable;
    FFNode(L&& lf) : callable(lf) {};

    void *svc(void *) {
        std::experimental::optional<TSout> ret;
        ret = callable();
        if(ret) {
            void *outslot = ff::ff_malloc(sizeof(TSout));
            TSout *out = new (outslot) TSout();
            *out = ret.value();
            return outslot;
        } else return {};
    }
};
```

way as GRPPI does: it represents an optional `TSout` data, meaning it can exists or not. If it does not exists, an empty object is returned instead of the `EOS` marker, which is subsequently handled by other overloading of the `FFNode` objects.

Note that this wrapper extends the generic `ff_node` instead of the typed one, because FastFlow prohibits to have a node object typed with a `void` input type and a specialized output type: in this case, it falls back to the generic version. Also, streaming patterns built using these wrappers make use of FastFlow's internal allocator, which is highly optimized for *producer/consumer* scenarios, and helps maintaining a low memory footprint during program execution.

An intermediate stage is modeled upon the `ff_node_t` object, and models a sequential stage that receives data from the input channel, executes its callable on the input data — if exists — and returns its result on the output channel. Both input and output data type have to be specified (see Listing 4.7), which are deduced at pattern-implementation level using C++ type support features.

A final stage does not return any data: it traces a consumer stage, where input data is stored or printed, and nothing is further placed in the data stream. It is modeled upon the `ff_node_t` object, and no output data type needs to be specified (see listing 4.8): if an input value exists, it is consumed and then destroyed (such that memory leaks are avoided), otherwise FastFlow's `GO_ON` marker is returned.

We built the back-end for stream parallel patterns using these building blocks. By exploiting template overloading, a Pipeline is constructed out of the stages passed to the variadic parameter in the Pipeline interface (see listing 4.1). Except for the very first stage, which must be the stream generator, other stages can be any intermediate sequential stage, or another streaming pattern that embeds a parallel logic (e.g., a Farm pattern). For instance, an intermediate stage can be a nested Pipeline as well.

At the base of this mechanism there is FastFlow's Pipeline pattern: by wrap-

Listing 4.7: FFNode for streaming patterns (intermediate)

```
template <typename TSin, typename TSout, typename F>
struct FFNode : ff_node_t<TSin,TSout> {
    F callable;
    FFNode(L&& lf) : callable(lf) {};

    TSout *svc(TSin *t) {
        std::experimental::optional<TSin> check;
        check = *t;
        if(check) {
            TSout *out = (TSout*) ff::ff_malloc(sizeof(TSout));
            *out = callable(check.value());
            return out;
        } else return {};
    }
};
```

Listing 4.8: FFNode for streaming patterns (final)

```
template <typename TSin, typename F>
struct FFNode<TSin,void,L> : ff_node_t<TSin,void> {
    F callable;
    FFNode(L&& lf) : callable(lf) {};
    void *svc(TSin *t) {
        std::experimental::optional<TSin> check;
        check = *t;

        if(check) {
            callable(check.value());
            t->~TSin();
            ff::ff_free(t);
        }
        return GO_ON;
    }
};
```

ping the `ff_pipeline` object, pattern matching on function templates allows to add stages to the Pipeline in the same order as they have been passed to the variadic parameter in the interface. The only requirement is that each stage must subclass one of the `ff_node` or `ff_node_t` objects, just like the `FFNode` does, as well as FastFlow's Farm pattern. Once the pipeline is constructed, the execution is triggered calling the `run_and_wait_end(void)` method, which starts the computation and awaits its termination in a cooperative way (see listing 4.9).

In order to match the nesting of a stream parallel pattern into the Pipeline, proper type checking is performed at compile time via template guards, and then the pattern is constructed and added to the Pipeline. In listing 4.9 there are two functions that insert stages into the underlying pipeline object: `add2pipe` actually adds the stage, while `add2pipeall` forwards additional stages passed via the variadic parameter, and recurs over the template functions until it finds the right

Listing 4.9: Construction of the Pipeline

```
template <typename Generator, typename ... Transformers>
void parallel_execution_ff::pipeline(Generator&& generate_op,
    Transformers&& ... transform_ops) const {
    ff_wrap_pipeline pipe(concurrency_degree_,
        generate_op,
        std::forward<Transformers>(transform_ops)...
    );
    pipe.run_and_wait_end();
}

class ff_wrap_pipeline : public ff::ff_pipeline {
public:
    template<typename Generator, typename... Transformers>
    ff_wrap_pipeline(size_t nw,
        Generator&& gen_func, Transformers&&...stages_ops)
        : nworkers{nw} {
        //type deduction...

        auto n = new ff::FFNode<void,generator_type,Generator>(
                                    std::forward<Generator>(gen_func) );

        add2pipe(n);                  // add first stage
        add2pipeall<generator_type>( // other stages
            std::forward<Transformers>(stages_ops)... );
    }
    // ...
};
```

match.

Listing 4.10 shows the creation of a Farm stage, which is added to the Pipeline before subsequent stages: after type deduction is performed, the actual workers of the Farm are constructed, on top of the `FFNode` object. The workers will be responsible for actually apply Farm's transformation over the input data stream, whose callable is captured at compile time by the `FarmTransformer` parameter. The workers are then added to an ordered Farm (`ff_OFarm`), which also adds default emitter and collector to the Farm. The new pattern is added to the Pipeline, while subsequent stages will be recursively added by matching the proper overloading.

Stream reduce and Stream filter are basically Farm patterns, and can easily be nested as Pipeline stages. Due to their semantics, they cannot be implemented as generic Farms, where workers carry on their job and return results, but need to be properly tuned.

The Stream reduce pattern aims at collapsing items appearing on the input stream, and then delivers these results to the output stream. This computation is applied to a subset of the input stream called *window*, where each window is processed independently. In order to accommodate such behavior, a simple Windowed Farm is created, where the emitter is responsible for buffering items coming from the stream, and dispatching them in batch to the workers, as soon as the required

Listing 4.10: Adding a Farm stage to the Pipeline

```
template <typename Input, typename FarmTransformer,
template <typename> class Farm,
typename ... OtherTransformers,
requires_farm<Farm<FarmTransformer>> = 0>
auto add2pipeall( Farm<FarmTransformer> && farm_obj,
    OtherTransformers && ... other_transform_ops) {
    // type deduction and correctness assertion...
    std::vector<std::unique_ptr<ff::ff_node>> w;
    for(int i=0; i<nworkers; ++i)
        w.push_back(
            make_unique<FFNode<Input,out_type,Farm<FarmTransformer>>>(
                std::forward<Farm<FarmTransformer>>(farm_obj)
            )
        );

    ff::ff_OFarm<Input,output_type> * theFarm =
        new ff::ff_OFarm<Input,output_type>(std::move(w));

    add2pipe(theFarm);
    add2pipeall<Input>(std::forward<OtherTransformers>(
                                    other_transform_ops)... );
}
```

window size is reached.

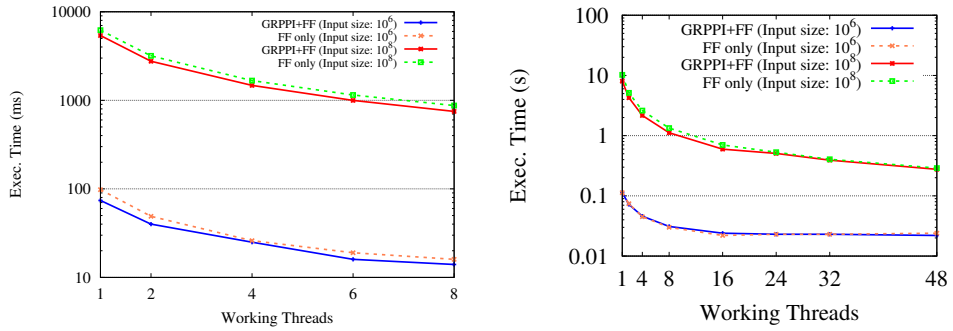
A Windowed Farm may have an *offset* parameter, which may create overlapping regions among two windows. This aspect is also managed by the emitter, that drops items or keeps duplicates, according to what scenario the offset size generates (depending on whether the offset is smaller, equal or greater than the window size). Again, in constructing the Windowed Farm, care has been taken for properly dealing with optional values, which are used to denote the end-of-stream event.

The Stream filter pattern applies a filter over the items in the input stream, where only those items satisfying the filter pass to the output stream. The filter is computed in parallel in a Farm fashion: despite rather straightforward, an *ad hoc* Farm has been created to reproduce this behavior in FastFlow. The reason lies in the mechanism FastFlow uses to communicate among the actors in play, which would fail to forward those items that satisfied filter's condition.

The Stream iteration pattern applies a transformation to a data item in a loop fashion, until a given condition is satisfied. When the condition is met, the result is sent to the output stream. This pattern has been implemented using a `ff_OFarm` pattern, whose workers are specialized `FFNode` objects that evaluate the condition and apply the transformation to the input items.

Task Parallel Patterns

GrPPI provides a Divide-and-Conquer (DAC) pattern, which applies a multi-branched recursion to break down a problem into several sub-problems, until the base case is reached: at this point distinct sub-problems can be solved in parallel



(a) Map pattern on *Paranoia*.

(b) Map pattern on *Occam*.

Figure 4.1: Map benchmark comparing execution times on both tested architectures.

and their solutions merged to form the definitive one. GRPPI comes with two interfaces for calling the DAC pattern, where one of them permits to explicitly pass the Predicate function that leads the divide phase, rather than implicitly use it in the Divide function body.

FastFlow provides a DAC pattern, whose interface requires the Predicate condition for the base case to be explicitly passed as a pure function: this perfectly matches GRPPI, thus the implementation is rather easy and only requires proper deduction of the output type.

4.3 FastFlow back-end: evaluation

The evaluation has been carried out on two machines, in order to test patterns' scalability and better compare the wrapped solution against the pure FastFlow implementation. The first machine is a server platform named *Paranoia*, equipped with an Intel Xeon *Ivy Bridge* E5-2650 v2 @2.60 GHz featuring 8 HyperThreading-enabled cores, 20 MB of L3 cache and 64 GB of RAM. The OS is a Linux Ubuntu 16.04.3 running kernel 4.4.0-97, where the C++ compiler used to assemble the whole project is GCC 6.3.0.

The second machine is part of the *OCCAM* HPC infrastructure located at the University of Torino, Italy. It is a NUMA machine featuring 4 sockets, each providing 12 HyperThreaded Intel Xeon E7-4830 v3 @2.10 GHz, summing up to 48 physical cores. It has 30 MB of L3 cache and 792 GB of RAM. Tests on this machine have been executed within a Docker container, running Linux Ubuntu 16.04.3 with kernel 3.10.0-514, where the C++ compiler used to assemble the whole project is GCC 6.3.0.

By default GRPPI enables two execution models: sequential (SEQ) and C++ Threads (NATIVE). All other back-ends have to be enabled at compile time by activating a specific variable (e.g., the flag `-DGRPPI_TBB_ENABLE=ON` enables

the use of Intel TBB). In the same way, FastFlow back-end is enabled at compile-time by passing the variable `-DGRPPI_FF_ENABLE=ON` to the *CMake* engine: this permits to use the `parallel_execution_ff` execution type, meaning that every pattern will by run using FastFlow back-end.

When this flag is enabled, CMake will check for the existence of FastFlow library in the project's build path. If it cannot find it, it will download the library and set up proper variables for correctly including header files when compiling the source code[1].

Each pattern is tested with two increasing input sizes, with increasing parallelism degree, up to the number of available physical cores. Every single test is run for ten times, and the average execution time is considered.

In order to perform the evaluation, we prepared sample benchmarks for some of the pattern described above, namely: *1)* Map, *2)* Reduce, *2)* Stencil and *4)* Farm. These benchmarks have been evaluated using either GrPPI with FastFlow back-end, or with a pure FastFlow implementation, in order to detect and analyze possible overheads induced by the wrapping.

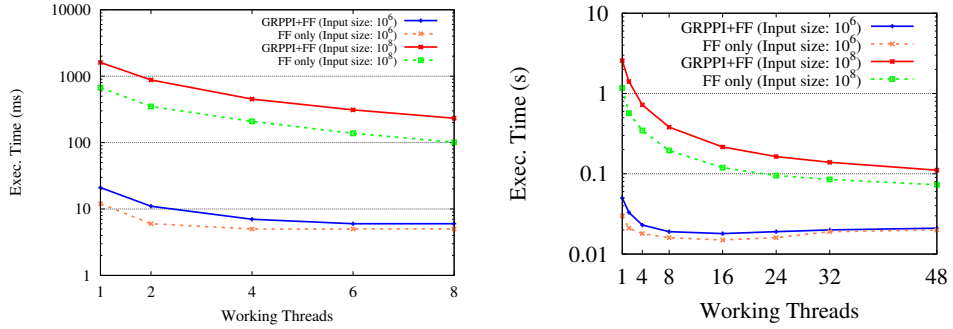
Map benchmark This benchmark has been used to compute a *daxpy* operation, that is, it computes the operation: $y = \alpha * x + y$, where x and y are vectors of floating point numbers, and α is a random floating point coefficient. The result overwrites the initial values of vector y. Figure 4.1 shows a comparison between the execution times on both architectures, obtained using GrPPI with FastFlow interface, and pure FastFlow implementation. Lines are coupled by input size, while figure 4.1a refers to the execution on *Paranoia*, and figure 4.1b refers to the execution on *OCCAM*.

Execution times result much similar, and scalability trends are also quite the same for both versions, on both machines. The GrPPI version seems to perform slightly better, which is probably due to better compiler optimizations facilitated by the extensive use of template classes in GrPPI.

Reduce benchmark This benchmark stresses the pattern by computing the addition of a sequence of natural numbers. Figure 4.2 shows a comparison between the two versions. Here, The pure FastFlow implementation also exhibits a similar behavior on smaller input datasets, while on the bigger dataset the it outperforms the other version. The overhead is still very tiny.

Stencil benchmark This benchmark performs a stencil computation over two one-dimensional vectors of the same size (v_1, v_2). The stencil function takes into account the two preceding and the two following neighbors of each item in each vector, and add their values to each item of the first vector. Figure 4.3 shows that on the NUMA machine, the execution time decreases more slowly when the concurrency degree exceedes 16 threads: this is likely due to the NUMA effect,

[1]Note that FastFlow is a header-only framework, and does not require any linking step.

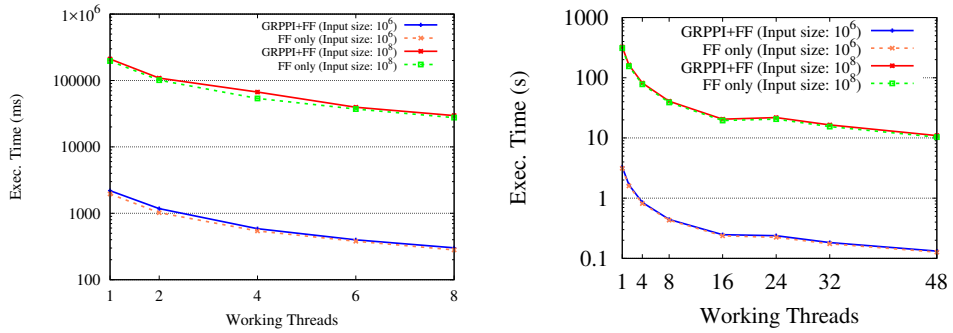


(a) Reduce pattern on *Paranoia*.

(b) Reduce pattern on *Occam*.

Figure 4.2: Reduce benchmark comparing execution times on both tested architectures.

where a thread that accesses memory on remote NUMA node pays higher transfer costs.



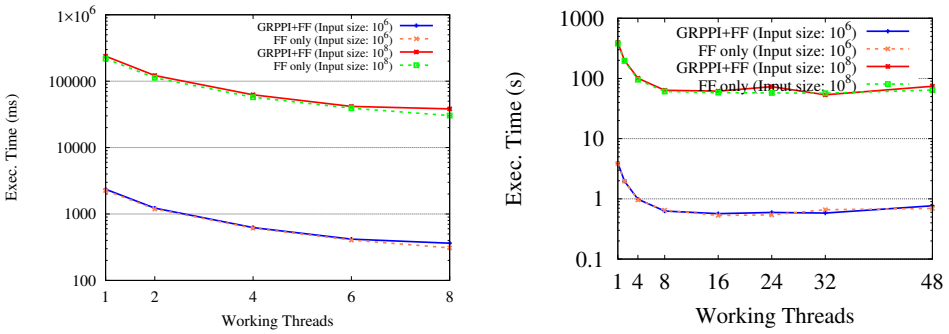
(a) Stencil pattern on *Paranoia*.

(b) Stencil pattern on *Occam*.

Figure 4.3: Stencil benchmark comparing execution times on both tested architectures.

Farm benchmark This benchmark executes some calculation on a stream of natural numbers: for each item in the input stream, workers perform repeated math operations on the input item, and return the result of their computation. As explained above, GRPPI builds all stream parallel patterns as a composition of the Pipeline pattern: for example, a Farm is a three-stages pipeline where the middle stage is a Farm pattern that exploits as many worker threads as the concurrency degree defines. In this comparison, the Farm pattern used with the pure FastFlow benchmark is actually a Pipeline+Farm composition, as reported in listing 4.11.

The Farm execution time appears to flatten with more than 8 workers, while it scales almost linearly before. Also, we were expecting better performance with



(a) Farm pattern on *Paranoia*.

(b) Farm pattern on *Occam*.

Figure 4.4: Farm benchmark comparing execution times on both tested architectures.

pure FastFlow implementations, because the GRPPI wrapping inevitably introduces an additional layer on top of the FastFlow API. Template classes used in GRPPI leverage compile time optimizations, that likely help to alleviate this problem, thus resulting in comparable performances with little or no overhead.

Listing 4.11: Farm patterns

```
// Pure FastFlow version
void farm_bench_ff(int cores) {
    ff::ff_Pipe<> pipe(make_unique<generator>(stream_len),
            make_unique<ff::ff_OFarm<long>>(worker_func, num_workers),
            make_unique<Collector>()
    );
    pipe.run_and_wait_end();
}

// GRPPI version
void farm_bench_grppi(const dynamic_execution & e, int cores) {
    grppi::pipeline(e,
        [&]() -> experimental::optional<long> {
            if (idx_in < v.size()) {
                idx_in++;
                return v[idx_in-1];
            } else
                return {};
        },
        grppi::farm(cores, // actually, uses policy's conc_degree
            [&](long x) {
                long res = do_work(x);
                return res;
        }),
        [&](long x) {
            output += x;
        }
    );
}
```

5. Advanced Patterns in FastFlow

In this part we describe the implementation of the advanced patterns in the native FastFlow parallel programming framework. The implementation provides full support of the entire set of the advanced patterns in the project.

5.1 High-level Patterns Interface

In this section, we describe the native FastFlow implementation of six advanced **RePhrase** patterns (Stream-Iterator, Stream Splitter, Stream Merger, Pool Windowed-Farm and Keyed-Farm).

5.1.1 Pool

The pool pattern models the evolution of a population of individuals. Iteratively, selected individuals are subject to evolution steps. The resulting new individuals are inserted in the population or discarded according to their score (*fitness*). The process is iterated up to a given number of iterations or to a given termination point.

The pattern has been implemented in two flavors: *synchronous* or *asynchronous* depending on how the different phases are concurrently executed. This results in two distinct implementations with slightly different semantics.

Synchronous Pool Interface

In this version, iterations behave as barriers, that is all the evolution and scoring computations are completed before considering the next iteration on the new population.

In the pattern, a "candidate selection" function (s) selects a subset of objects belonging to an unstructured object pool (P). The selected objects are processed by means of an "evolution" function (e). The set of objects computed by the evolution function on the selected object are filtered through a "filter" function (f) and eventually inserted into the object pool. At any insertion/extraction into/from the object pool a "termination" function (t) is evaluated on the object pool to determine whether the evolution process should be stopped or continued for further iterations. The computation takes as an input a population $x_0, \ldots, x_k$ of individuals and computes in parallel the following function:

function POOL(P)
  while $not(t(P))$ **do**
    $N = e(s(P))$
    $P = (P \setminus s(P)) \cup f(N, P)$x
  end while
end function

Listing 5.1 shows the interface of the pattern. The Synchronous version of the Pool pattern is implemented by the class `PoolEvolution`. The programmer has to provide the required functions to the pool constructor.

Listing 5.1: Synchronous Pool Evolution.

```
template <typename T, typename env_t=char>

typedef void    (*selection_t)  (ParallelForReduce<T> &,
std::vector<T> &, std::vector<T> &, env_t &);
typedef const T& (*evolution_t)  (T&, const env_t&, const int);
typedef void     (*filtering_t)  (ParallelForReduce<T> &,
std::vector<T> &, std::vector<T> &, env_t &);
typedef bool     (*termination_t)(const std::vector<T> &pop, env_t &);

poolEvolution (size_t maxp,        // maximum parallelism degree
std::vector<T> & pop,      // the initial population
selection_t sel,       // the selection function
evolution_t evol,      // the evolution function
filtering_t fil,       // the filter function
termination_t term,        // the termination function
const env_t &E= env_t(), bool spinWait=true)
```

The selection and filtering phases can be performed in parallel by exploiting a `ParallelForReduce` pattern (i.e. the programmer can defines their implementation by exploiting a `ParallelForReduce` object instantiated and maintained by the `PoolEvolution` class). Evolutions on distinct objects are performed in parallel.

The `env_t` type passed as template parameter is the type representing the “external environment" of the pool pattern. It is the type of a data-structure passed in the constructors, and accessed in a read-only way in the evolution phase, which is used to store “global" information between the computation of the four different phases of the pattern.

Synchronous Pool example

Listing 5.2 shows a usage example of the synchronous version the pattern.

In this case, the population is composed by integer numbers that are evolved by increasing their value. At each evolution step, at most K elements are removed from the population. The evolution continues as long as we have odd numbers.

Listing 5.2: `poolEvolution` usage example.

```
#define MAXMUTATIONS  4
#define K             2

struct Element {
    Element(size_t n=0): number(n),nmutations(0) {}
    size_t number, nmutations;
};

// if we have at least an odd element, than we go on
bool termination(const std::vector<Element> &P, poolEvolution<Element>::
   (cont.)envT&) {
    for(size_t i=0;i<P.size(); ++i)
        if (P[i].nmutations < MAXMUTATIONS && P[i].number & 0x1) return
           (cont.)false;
    return true;
}

// selects the first half of the population
void selection(ParallelForReduce<Element> &, std::vector<Element> &P,
               std::vector<Element> &output, poolEvolution<Element>::envT
                  (cont.)&) {
    for(size_t i=0;i<P.size()/2;++i)
        output.push_back(P[i]);
}

const Element& evolution(Element & individual, const poolEvolution<Element
   (cont.)>::envT&,const int) {
    individual.number += decltype(individual.number)(individual.number/2);
    individual.nmutations +=1;
    return individual;
}

// remove at most K elements randomly
void filter(ParallelForReduce<Element> &, std::vector<Element> &P,
            std::vector<Element> &output, poolEvolution<Element>::envT&) {
    if (P.size()<K) { output.clear(); return; }
    output.clear();
    output.insert(output.begin(),P.begin(), P.end());

    for(size_t i=0;i<K;++i) {
        auto r = random() % (output.size()); // MA Changed from P.size()
           (cont.)to output.size();
        output.erase(output.begin()+r);
    }
}

int main(int argc, char* argv[]) {
    //...
    //....build the population....
    std::vector<Element> P;
    buildPopulation(P, size);

    poolEvolution<Element> pool(nw, P,selection,evolution,filter,
       (cont.)termination);
    pool.run_and_wait_end();

    //use the final popuation
    //...
}
```

Asynchronous Pool Interface

In this version, individual results of the evolution process are immediately evaluated and, in case, inserted in the population asynchronously with respect to the processing of other individuals/evolutions. The pattern takes in input:

- a population of individuals P;
- an evolution function e;
- a scoring function f;
- a termination condition t;

With respect to the Synchronous Pool Evolution pattern, in this implementation the selection function (s) returns the top-K individuals according to their scores.

The operating behavior of the pattern is the following one:

1. at initialization time the best K candidates of the population are selected. This requires to compute (possibly in parallel) their scoring using f;
2. initially, these top-K individuals start the evolutionary process. For each of them we compute the evolution and the scoring functions;
3. then, every time that a new evolved individual is available, it is inserted in the current remaining population. The best one is selected and starts the evolutionary process;
4. every time that a new evolved element is available, we evaluate the termination function t. If it is true (or if we reach a maximum number of iterations) the computation ends.

Listing 5.3 illustrates the interface of the pattern. The pattern implementation is realized by the class `ff_PoolEvolutionAsynch`.

Listing 5.3: Asynchronous Pool Evolution.

```
template<typename T>

ff_PoolEvolutionAsynch(std::vector<T> &population,
const std::function<double(const T&)> &scoring,
const std::function<void(T &)> evolve,
const std::function<bool(PoolIterator, PoolIterator)> &termination, int k
   (cont.), long max_iterations,int nw)
```

The programmer has to pass all the required functions, the starting population and other additional info such as the number of elements `k` to select, the maximum number of iterations and the parallelism degree (`nw`).

Listing 5.4: `ff_PoolEvolutionAsynch` usage example.

```
bool termination(ff_PoolEvolutionAsynch<int>::PoolIterator begin,
   (cont.)ff_PoolEvolutionAsynch<int>::PoolIterator end)
{
    bool terminate=true;
    double avg=0;
    int size=end-begin;
    while(begin!=end){
        avg+=(*begin)->get_score();
        begin++;
    }
    avg/=size;
    return avg>10;
}

int main()
{
    //initialize population and define the pool parameters
    std::vector<int> pop={1,2,3,4,5,6,7,8,9,10};
    int k=3;
    int max_iterations=1000;
    int num_workers=2;
    ff_PoolEvolutionAsynch<int> pool(pop,
                                     [](const int &e){return e+1;},
                                     [](int &i){i++;},termination,k,
                                        (cont.)max_iterations,num_workers);
    pool.run_and_wait_end();

    //use the final population
    //...
}
```

Asynchronous Pool example

Listing 5.4 shows a simple example of use of the asynchronous version the pattern.

In this case the population is composed by integer numbers, which are "evolved" by incrementing their value by one. The termination conditions checks that the average value of the population is over a given value

5.1.2 Stream-Iterator

The pattern iterates the computation of another pattern over one or more items appearing onto the input stream. With respect to the StreamIterator pattern presented in Deliverable 2.4, here there is the possibility to specify an additional boolean predicate *guard*. It tells whether the item computed by the inner pattern should also be output on the output stream in case the *predicate* tells it has to be processed again.

Interface

The pattern is implemented by class `ff_SI_MO` and its interface is described in Listing 5.5.

Listing 5.5: `ff_SI_MO` FastFlow interface.

```
template<typename IN_T>
ff_SI_MO(ff_node &nested, const std::function<bool (const IN_T&)>
   (cont.)predicate, const std::function<bool (const IN_T&)> guard,
   (cont.)SIPriority priority=SIPriority::BALANCED )
```

The pattern works on a stream of type `IN_T`. The parameters that must be provided are:

- the *nested* pattern to iterate. Every FastFlow pattern can be passed, provided that it has a $1 : 1$ selectivity, that is for every input element one output element is produced;
- a `predicate` stating whether the result of the nested pattern must be flown again to the nested pattern input stream;
- a *guard* output predicate. It is true if an element that must be re-flown must also be sent to the output (by copying it);
- optionally, the programmer can indicate a priority for the scheduling policy of input elements. In particular it can decide that:
  - elements arriving to the `ff_SI_MO` input stream and elements that are going to be flown again to the nested pattern are treated with the same priority. This is the default behavior, signaled with the `SIPriority::BALANCED` flag;
  - elements that are flown again to the nested pattern have higher priority w.r.t. elements coming from the input stream. This is signaled using the `SIPriority::NESTED_PATTERN` flag.

Example

Listing 5.6 shows a simple example of use of the pattern.

In this case the nested pattern is a single sequential `ff_node`, that receives a stream of `long` input elements, increments them and send the result in the output stream. The predicate passed to the `ff_SI_MO` pattern impose that the stream elements have to be flown again if they are multiple of 2 or 3. The output guard, states that elements multiple of 5 must be also sent in output. In addition, in the pattern declaration we indicate to give more priority to re-flown elements with respect to the input stream elements.

Listing 5.6: `ff_SI_MO` usage example.

```
 class NestedPattern:public ff_node_t<long>{
    public:
    long *svc(long *t) {
        *t=*t+1;    //increment the input element
        return t;
    }
};

int main(){
    //...
    NestedPattern nested_pattern;
    ff_SI_MO<long> si(nested_pattern,[](const long &t){
            return((t%2)==0 || (t%3)==0);
        },[](const long &t){
            return ((t%5)==0);
        }, ff_SI_MO<long>::SIPriority::NESTED_PATTERN);
    ...
}
```

5.1.3 Windowed-Farm

As stated in the previous chapter for the GRPPI implementation, the Windowed-Farm pattern dynamically builds windows of the input stream, that is bounded regions of the input stream composed of a set of consecutive input items that are logically grouped according to a *cout-based* or a *time-based* policy. Basically, the pattern goes over the items in each group and produces an output result per window which is delivered onto the stream out of the pattern.

Interface

In the FastFlow native implemenation the Windowed-Farm is provided in two different interfaces supporting two classes of sliding-window streaming queries:

- *non-incremental interface*: such interface is applied when the available algorithm (business logic code provided by the user during the pattern instantiation) assumes to have the whole set of input items composing the same window. In other words, such algorithm goes over the items of the window and produces the corresponding window result, i.e. the user-defined function is of type $\mathcal{F} : \alpha$ collection $\rightarrow \beta$ where α is the type of the generic input item and β of the output result data type;

- *incremental interface*: there are several notable cases where the window result can be computed without waiting to have the whole bulk of window items. Instead, the results can be incrementally updated as soon as a new input item belonging to the window arrives from the input stream and it is ready to be processed. The user-definied function is of type $\mathcal{F} : \alpha \times \beta \rightarrow \beta$

where α is the type of the generic input item and β of the output result data type.

The choice of the interface to use is a degree of freedom of the user, i.e. the pattern implementation is provided for both the cases, and the correct implementation is statically selected according to the type of constructor utilized by the user to build the pattern and to fill its internal business logic. The pattern interface is shown in Listing 5.7.

Listing 5.7: Windowed-Farm interface.

```
template <typename tuple_t, typename result_t>
Win_Farm(f_winfunction_t F, size_t win_len, size_t _slide_len,
         win_type_t win, size_t n);
```

The first template argument is the type α of the input item while the second is the type β of the output result. The type of the output result must be default constructible. Other template arguments (not shown) are available to control advanced internal features of the pattern such as the container used to store the input items of the windows. The pattern constructor takes five arguments listed below:

- the first argument is the function F to be applied over each window of the input stream. As previously discussed, depending on the singature of the provided function, the non-incremental or the incremental implementation of the pattern is properly instantiated;
- the second and the third argument are the window length and the sliding factor. In case of count-based windows both the parameters are expressed in number of input items, in case of time-based windows instead they are expressed in time units. A mixing of the two combinations (i.e. the so called slide-by-tuple windows) are not supported in the current release since they represent very specific window instantiation that we considered of poor interest for the project;
- the fourth parameter is an enumeration with two possible values: `CB` for count-based windows or `TB` for count-based windows;
- the last input parameter is the concurrency degree utilized internally by the pattern. In case of $n = 1$, the pattern executes all the sliding windows sequentially. With $n > 1$ up to n windows can be executed in parallel by multiple threads depending on the stream arrival rate. In every case the pattern emits the output results in order of the corresponding window identifier (starting from zero).

Example

Listing 5.8 shows an example of the Windowed-Farm in FastFlow instantiated to solve a very simplified benchmark query where the input items are objects of

Listing 5.8: Usage example of the Windowed-Farm pattern with the non-incremental interface.

```
using const_iterator = Win_Farm<tuple_t, result_t>::const_iterator;
// non-incremental window function
auto F = [](size_t key, size_t wid, const_iterator it,
            const_iterator end, result_t &r) {
              r.sum = 0;
          while (it != end) {
         result += (*it).value;
         it++;
          }
        return 0;
};
// creation of the Win_Farm pattern
Win_Farm<tuple_t, result_t> wf(F, win_len, win_slide, CB, pardegree);
// creation of the pipeline
Generator generator(stream_len);
Consumer consumer();
ff_Pipe<tuple_t, result_t> pipe(generator, wf, consumer);
if (pipe.run_and_wait_end() < 0)
   return -1;
else
   return 0;
```

the type `tuple_t` and the computation goes over the input items and computes a stream of output results of type `result_t`, which contains the sum of all the items in the window. The computation is deliberately easy and should be considered as an example to understand the possible pattern instantiation using the FastFlow C++11 native interface.

The pattern stores the input items in an internal container whose type can be optionally passed as an additional template argument to the pattern. Function F is the non-incremental window function that takes as input the identifier of the sub-stream (denoted by an integer), the unique identifier of the window to be processed (starting from zero), the initial and the final iterators that will be used to access sequentially the window content, and the result of the window computation, i.e. an heap-based variable allocated by the FastFlow runtime (it can be any complex data structure depending on the computation to be processed). In the example the pattern is instantiated with count-based windows, while the generator is a FastFlow node in charge of generating a stream of items containing random integer values, while the consumer is a node responsible for receiving the window results by storing them in a output textual file. The example with time-based windows is straightforward.

Listing 5.9 shows the implementation of the same problem when the window function has an incremental nature:

As we can observe only the signature of the input function to the pattern is modified.

Listing 5.9: Usage example of the Windowed-Farm pattern with the incremental interface.

```
using const_iterator = Win_Farm<tuple_t, tuple_t>::const_iterator;
// incremental window function
auto F = [](size_t key, size_t wid, tuple_t item, tuple_t &r) {
   r.sum += item.value;
   return 0;
};
// creation of the Win_Farm pattern
Win_Farm<tuple_t, result_t> wf(F, win_len, win_slide, CB, pardegree);
// creation of the pipeline
Generator generator(stream_len);
Consumer consumer();
ff_Pipe<tuple_t, result_t> pipe(generator, wf, consumer);
if (pipe.run_and_wait_end() < 0)
   return -1;
else
   return 0;
```

5.1.4 Keyed-Farm

The Keyed-Farm assumes that the input stream conveys items belonging to different logical sub-streams. All the items belonging to the same sub-stream are marked with a unique *key* identifier. The idea of this pattern is to apply the sliding-window processing paradigm on different sub-streams. According to the sliding-window policy (e.g., count-based or time-based), different windows can be identified per sub-stream and a user-defined function is applied on each window and produces an output result to be delivered onto the stream out from the pattern. In other words, if the windowing semantics states that the window length is of $1,000$ items and the sliding factor is of 100 items, a new window is triggered every time new 100 items with the same key are received, and the processing function goes over the last $1,000$ items of that key that were previously received.

In the parallel execution semantics, the patterns exploit parallelism among different logical sub-streams, i.e. the windows of the same sub-stream are rigorously executed sequentially in-order (from the first window to the last one), while windows of different sub-streams can be executed in parallel by multiple implementation threads.

Interface

Also for the Keyed-Farm pattern, the native FastFlow implementation provides the non-incremental and the incremental interface depending on the type of user-defined function provided by the high-level programmer during the pattern instatiation. The generic template definition is shown in Listing 5.10:

Listing 5.10: Keyed-Farm interface.

```
template <typename tuple_t, typename result_t>
Key_Farm(f_winfunction_t F, size_t win_len, size_t _slide_len, win_type_t
   (cont.) win, size_t n, f_routing_t routing);
```

The constructor of the pattern has a very similar interface of the Windowed-Farm pattern. In addition, the last input argument is the routing function that states the corresponding between the input item key attribute (an integer starting from zero) onto the corresponding worker identifier (FastFlow node in charge of executing all the windows of that key in order). If not provided, the default routing function assumes a round-robin correspondence, i.e. $k \mod n_w$, where k is the key attribute of the input item and n_w is the pattern's internal parallelism degree. It is worth noting that such pattern is completely useless from the point of view of the parallism exploitation in case of input streams convey items all having the same key attribute value.

Example

Similarly to the Windowed-Farm, we show in this part two examples of the Keyed-Farm pattern instantiated with the non-incremental and the incremental interface.

Also in this case we show a very simplified computation which consists in computing for each window of items withing the same logical sub-stream the item with the maximum value. The input type `tuple_t` is a struct with a field $value$ of type `int`, while the output result of type `result_t` is a struct with a field id corresponding with the unique identifier of the item with the maximum value and $value$ its value. The pattern is instantiated with count-based windows (the case of time-based windows require slight modification in the source code).

Listing 5.11 shows the first case with the pattern instantiated with the non-incremental query.

Also in this example the generator and the consumer nodes are responsible for generating a several multiplixed sub-streams and of recording the output results in a file respectively. Listing 5.12 shows the same application where the pattern has been instantiated with the incremental interface.

5.2 Evaluation of High-level Patterns

In this part, we propose a brief evaluation of the advanced patterns as they are implemented in the native FastFlow interface.

- *Target platform.* Dual-CPU Intel Sandy-Bridge multicore with 16 hyper-threaded cores operating at 2GHz with 32GB or RAM. Each core has a private L1d (32KB) and L2 (256KB) cache. Each CPU is equipped with a shared L3 cache of 20MB. Linux 3.14.49.

- *Software.* The code of the experiments has been compiled with the `gcc` compiler (version 7.2.0) and the `-O3` optimization flag.

Listing 5.11: Usage example of the Keyed-Farm pattern with the non-incremental interface.

```
using const_iterator = Key_Farm<tuple_t, result_t>::const_iterator;
// non-incremental window function
auto F = [](size_t key, size_t wid, const_iterator it, const_iterator end
   (cont.), result_t &r) {
    r.value = 0;
    while (it != end) {
       if ((*it).value > r.value) {
           r.value = (*it).value;
           r.id =  (*it).id;
       }
       it++;
    }
    return 0;
};
// creation of the Key_Farm pattern
Key_Farm<tuple_t, result_t> kf(F, win_len, win_slide, CB, pardegree);
// creation of the pipeline
Generator generator(stream_len);
Consumer consumer();
ff_Pipe<tuple_t, result_t> pipe(generator, kf, consumer);
if (pipe.run_and_wait_end() < 0)
   return -1;
else
   return 0;
```

Listing 5.12: Usage example of the Keyed-Farm pattern with the incremental interface.

```
using const_iterator = Key_Farm<tuple_t, tuple_t>::const_iterator;
// incremental window function
auto F = [](size_t key, size_t wid, tuple_t item, tuple_t &r) {
   if(r.value > item.value) {
       r.value = item.value;
       r.id = item.id;
   }
   return 0;
};
// creation of the Win_Farm pattern
Key_Farm<tuple_t, result_t> kf(F, win_len, win_slide, CB, pardegree);
// creation of the pipeline
Generator generator(stream_len);
Consumer consumer();
ff_Pipe<tuple_t, result_t> pipe(generator, kf, consumer);
if (pipe.run_and_wait_end() < 0)
   return -1;
else
   return 0;
```

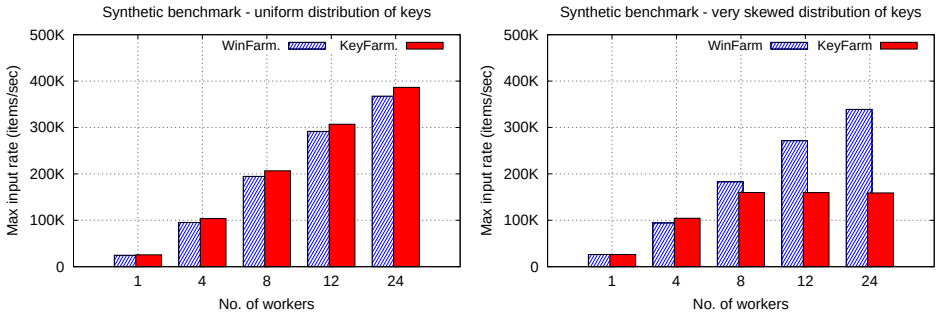
- The Windowed-Farm and Keyed-Farm patterns have been tested on a financial application [9] that computes a suite of complex statistical aggregates used for algotrading tasks. The input stream conveys items belonging to $1,000$ different sub-streams and the patterns are instantiated with count-based windows with length $1,000$ and slide 200 items.

5.2.1 Performance analysis of the Windowed-Farm and the Keyed-Farm patterns

For each parallelism degree we determine the highest input rate (throughput) sustainable by the parallel pattern before becoming a bottleneck. To detect it, we repeat the experiments several times with growing input rates. The generator checks the TCP buffer of the socket. If it is full for enough time, it stops the computation and the last sustained rate is recorded. Figs. 5.1a and 5.1b show the results in two scenarios:

- in the first one, the input stream conveys items belonging to thousands of distinct keys that are uniformly distributed;
- in the second one, we use a very *skewed* key distribution where $p^{max} = 16\%$ of the overall input items belong to the same key (the most frequent one).

We report the results with two worker threads per core (24), which is the best hyperthreaded configuration found in our experimental setting. In the first scenario shown in Fig. 5.1a, the Keyed-Farm pattern with the default routing function provides slightly higher throughput than Windowed-Farm due to a lower overhead in the scheduling phase.



(a) Uniform distribution.

(b) Skewed distribution.

Figure 5.1: Maximum throughput measured for the Windowed-Farm and the Keyed-Farm patterns in a financial benchmark with count-based sliding windows.

Differently, in the second case with a very skewed key distribution (Fig. 5.1b), the thoughput achieved with the Keyed-Farm pattern is substantially lower than the maximum one achieved by the Windowed-Farm pattern. This is an expected

result, because the Windowed-Farm exploits parallelism between different windows (also belonging to the same sub-stream), while in the Keyed-Farm only windows of different keys can be executed in parallel. Therefore, in the latter case a (possibly statically impredicatable) skewed distribution of keys hampers the possibility to achieve a near-optimal scalability due to load imbalance.

5.3 Basic Building Blocks Interface

This final part of this chapter focuses on the native FastFlow implementation of the basic building block patterns as they were described in D2.5.

5.3.1 Stream Splitter

A Stream Splitter generates a number of output streams out of a single input stream, according to a parameter policy (i.e. *Round Robin*, *Random* or *User Defined*). The programmer has to declare the set of nodes to which the Splitter will send the different output streams (one for each node). The Stream Splitter (together with the Stream Merger) allows the user to define arbitrary computational graph.

Interface

The `ff_Splitter` class implements a Stream Splitter. It has three distinct constructors, each one capturing a different scheduling policy. Listing 5.13 shows the interface for the case in which a *Round Robin* policy is adopted. In this case the programmer has to pass only the set of nodes that will receive the data from the splitter.

Listing 5.13: Stream Splitter interface: Round Robin policy.

```
template <typename IN_T>
ff_Splitter(std::vector<ff_node*> & nodes)
```

In Listing 5.14, the constructor for the case of the *Random* scheduling policy is shown. In this case the programmer has to pass also an `unsigned int` that will be used as seed for the random number generator used to select the data receiver.

Listing 5.14: Stream Splitter interface: Random policy.

```
template <typename IN_T>
ff_Splitter(std::vector<ff_node*> & nodes, unsigned int seed)
```

Finally, Listing 5.15 shows the case of a Splitter with a *User Defined* scheduling policy. In this case the programmer must specify a function that, given an input

stream element, returns the `id` of the stream (and, consequently the attached node) to which forward the element.

Listing 5.15: Stream Splitter interface: User Defined policy.

```
template <typename IN_T>
ff_Splitter(std::vector<ff_node*> & nodes,
const std::function<int(const IN_T&)> routing_fun)
```

Please note that:

- attached nodes are labelled with id starting from 0;
- the attached node must be started (and terminated) directly by the programmer using the `run()` and `wait()` methods of the `ff_node` class.

Example

Listing 5.16 shows a usage example of the Stream Splitter pattern. In this case the arriving elements are splitted to the different attached nodes by using a Round Robin strategy.

Listing 5.16: Stream Splitter Example.

```
int main()
{
    //...
    std::vector<ff::ff_node *> nodes;

    //...populates the vector of nodes to be attached to the splitter

    //create a Round Robin Splitter and a pipeline with a stream generator
    ff::ff_Splitter<int> splitter(nodes);
    ff::ff_Pipe<> pipe(stream_generator,splitter);

    //start the nodes using the run and wait methods
    for(ff_node *node: nodes )
        node->run();

    pipe.run_and_wait_end();

    //stop the nodes
    for(ff_node *node: nodes )
        node->wait();
}
```

5.3.2 Stream Merger

The Stream merger pattern accepts a number s of input streams and produces an output stream hosting the items appearing on the inputs, ordered according to a

given policy. Optionally, the programmer can specify a set of functions $f_1, f_2, \ldots, f_s$ such that f_i is applied to elements arriving from stream i before its delivery to the output stream.

Interface

The `ff_Merger` class implements a Stream Merger. It defines two constructor that can be used to specify two different merging policies. Listing 5.17 shows the case in which a *Time Ordered* merge is required, i.e. output items appear in the same order of arrival to the merger. In this case, the user has to specify the set of FastFlow nodes that will be attached to the merger and (optionally) the set of function to be applied on input elements.

Listing 5.17: Stream Merger interface: Time Ordered merge.

```
template <typename IN_T>
ff_Merger(std::vector<ff_node *> &nodes, const
std::vector<std::function<void(IN_T&)>> &funct_set={})
```

In Listing 5.18 it is shown the case in which a User Defined ordering is used, i.e. output items appearing at the same time at the merger inputs are sorted by a user "compare" function passed as argument.

Listing 5.18: Stream Merger interface: User Defined merge.

```
template <typename IN_T>
ff_Merger(std::vector<ff_node *> &nodes, const
std::function<bool(IN_T *, IN_T*)> cmp,const std::vector<std::function<
   (cont.)void(IN_T&)>> &funct_set={})
```

Please note that as in the case of the Stream Splitter the attached node must be started (and terminated) directly by the programmer using the `run()` and `wait()` methods of the `ff_node` class.

Example

Listing 5.19 shows an example of application of the Stream Merger pattern. A certain number of nodes are created each of them sending a stream of integer. The merger orders them according to a `cmp` function and applies a user defined function (for example, increments them).

Listing 5.19: Stream Merger usage example.

```
int main()
{
    //...

    std::vector<ff::ff_node *> nodes;
    //... populate the vector of attached nodes ...

    std::vector<std::function<void(int &)>> funs;
    //... populate the vector function ....

    //build the merger
    ff::ff_Merger<int> merger(nodi, [](int *a, int *b){return *a<*b;},funs
       (cont.));

    //start the nodes
    for(ff_node *node: nodes )
        node->run();

    //start the merger
    ff::ff_Pipe<> pipe(merger,visualizer_node);
    pipe.run_and_wait_end();
}
```

6. Software availability

The software discussed in this deliverable may be accessed as follows:

- the full GrPPI implementation is avaialable at the UC3M git repository:

  https://github.com/arcosuc3m/grppi

- the FastFlow backend for the GrPPI is currently available at the UNITO git repository:

  https://github.com/alpha-unito/grppi

  By the end of the project, it will be integrated into the main GrPPI git repository at UC3M.

- the final FastFlow RePhrase release, including all the extended patterns discussed in Chap. 5 are available at the UNIPI FastFlow web site:

  https://calvados.di.unipi.it/storage/software/rephrase/

All the software is released under different kind of open source licenses.

7. Conclusions and future work

In this deliverable, we have reviewed the state-of-the-art about and have presented the extension of GRPPI for supporting some of the patterns in the **RePhrase** advanced parallel pattern set. As demonstrated through the experimental evaluation, the use cases implemented with the proposed patterns attain remarkable speedup gains compared with their corresponding sequential versions. Although in some cases, the parallelism degree is limited by the pattern nature. We also proved that leveraging GRPPI reduces considerably the number of LOCs that have to be modified in the original codes to turn them parallel with respect to using the parallel frameworks directly. In general, we believe that these advanced patterns can eventually be incorporated in domain-specific applications so as to easily parallelize them, without having a deep understanding of existing parallel programming frameworks or third-party interfaces.

We also outlined the implementation of the patterns discussed in D2.1 using FastFlow, which differ from the one pesented by GRPPI, because *a)* FastFlow design and implementation is pre-existing w.r.t. GRPPI, and *b)* FastFlow has been designed with performance as the main goal rather than programmability "á la C++", and this led to quite different choices in terms of pattern API design. As a matter of fact, FastFlow uses *pointers* as capabilities moved from pattern to patterns or — at a lower level of abstraction — between different concurrent activities of the pattern implementation(s). By providing business logic code working on data pointers (rather than data values) the FastFlow pattern application programmers subscribe the idea ov moving capability associated to the pointer between concurrent activities. The framework guarantees that nothing goes wrong provided the capability concept is respected (correctly exploited) by the programmers. Moreover, the usage of plain C/C++ pointers has been enforced to support compact and extremely efficient implementation of the inter-concurrent activity (thread) communications, preventing the usage of more complex (and opaque) pointer implementations.

This notwithstanding, FastFlow patterns have been wrapped in GRPPI to test *a)* the feasibility of encapsulation of FastFlow after the other GRPPI implementations (namely, the one using C++ threads, the OpenMP one and the Intel TBB one), and *b)* the loss in terms of perfomance achieved when wrapping FastFlow into GRPPI. Results have been shown that demonstrate the feasibility of the wrapping

and a moderate (often negligible) performance decrease in all those cases where the pass-by-value semantics of GRPPI is outperformed by the pointer/capability semantics of FastFlow.

As future work, we plan to support other advanced parallel patterns in GRPPI, such as the *keyed stream farm*, *stream pool* and *image convolution*. Furthermore, we intend to include other execution environments as for the offered parallel frameworks, e.g., SkePU. An ultimate goal is to provide support for accelerators via CUDA Thrust and OpenCL SYCL.

Bibliography

[1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Pool evolution: A parallel pattern for evolutionary and symbolic computing. *International Journal of Parallel Programming*, 44(3):531–551, 2016.

[2] Marco Aldinucci, Marco Danelutto, Maurizio Drocco, Peter Kilpatrick, Guilherme Peretti Pezzi, and Massimo Torquati. The loop-of-stencil-reduce paradigm. In *Proc. of Intl. Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (RePara)*, pages 172–177, Helsinki, Finland, August 2015. IEEE.

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. FastFlow: high-level and efficient streaming on multi-core. In *in Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana*, page 13, 2012.

[4] Marco Aldinucci, Guilherme Peretti Pezzi, Maurizio Drocco, Concetto Spampinato, and Massimo Torquati. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *International Journal of High Performance Computing Application*, 2015.

[5] Saman Amarasinghe, Mary Hall, Richard Lethin, Keshav Pingali, Dan Quinlan, Vivek Sarkar, John Shalf, Robert Lucas, and Katherine Yelick. ASCR programming challenges for exascale computing. Technical report, U.S. DOE Office of Science (SC), 2011.

[6] Henrique C. M. Andrade, Bugra Gedik, and Deepak S. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, New York, NY, USA, 1st edition, 2014.

[7] Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Department of Computer Science, Heriot-Watt University, December 2013.

[8] Doina Bucur, Giovanni Iacca, Giovanni Squillero, and Alberto Tonda. *An Evolutionary Framework for Routing Protocol Analysis in Wireless Sensor Networks*, pages 1–11. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[9] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. Data stream processing via code annotations. *The Journal of Supercomputing*, Jun 2016.

[10] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, pages e4175–n/a, April 2017.

[11] Johan Enmyren and Christoph W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[12] Ewa Gajda-Zagórska. *Multiobjective Evolutionary Strategy for Finding Neighbourhoods of Pareto-optimal Solutions*, pages 112–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[13] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[14] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.