Project no. 644235

REPHRASE

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

Software for implementations of initial patterns
D2.4

Due date of deliverable: July 31st, 2016 (M16)

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

Responsible institution: UNIPI
Editor and editor's address: Marco Danelutto, UNIPI

Version 0.2 - August 31st, 2016

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	31/08/16	Marco Aldinucci	UNITO	First version
2	15/09/16	Manuel F. Dolz, J. Daniel Garcia, Marco Aldinucci	UC3M, UNITO	Fixed code snippets
3	26/09/16	Kevin Hammond	USTAN	Syntax fixes
4	20/02/17	Marco Danelutto	UNIPI	Added changelog

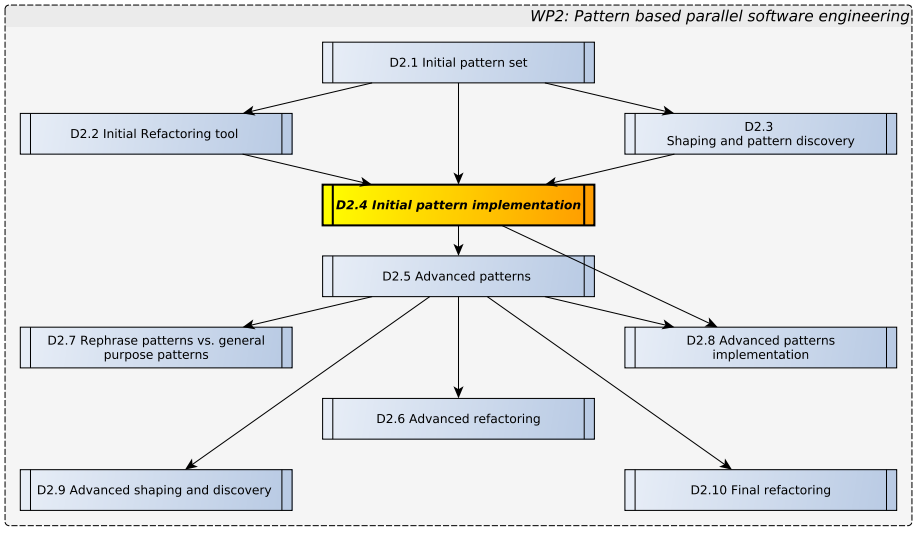
Executive Summary

This document is the fourth 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 initial patterns” as described in the amended DoW. In particular, D2.4 integrates the results of the a crucial phase of WP2 (T2.2 “Pattern implementation”) where, according to the amended DoW *we will implement an initial set of patterns identified in the first phase of T2.1, 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).*

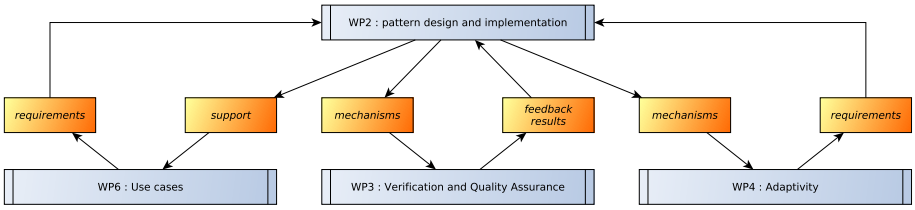
The deliverable details three different software contributions: *i)* a uniform C++11 pattern interface along with an implementation of the patterns in D2.1 on top of C++ threads, OpenMP, Intel TBB and Thrust, *ii)* a partial implementation of the C++ pattern interface API on top of FastFlow, outlining the pros and cons of the API implementation in terms of performance, and *iii)* a complete implementation of the D2.1 patterns in FastFlow, provided through standard FastFlow native interface.

The main contributions to this deliverable may be summarized as follows: FastFlow high level pattern (UNIPI), GrPPI (UC3M), GrPPI FastFlow interface (UNITO).

The placement of D2.4 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:


WP2 : pattern design and implementation
requirements
support
mechanisms
feedback
results
mechanisms
requirements
WP6 : Use cases
WP3 : Verification and Quality Assurance
WP4 : Adaptivity

Contents

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 [10]. 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, for writing applications in parallel, apart from the knowledge necessary in the application domain.

Approaches to relieve this burden are pattern-based parallel programming frameworks, such as SkePU [12], FastFlow [5] or Intel TBB [24]. In this sense, patterns provide a way to encapsulate (using a building blocks approach) algorithmic aspects, allowing users to implement robust, readable and portable solutions with such high-level abstractions. Basically, these patterns instantiate parallelism while hide away the complexity of concurrency mechanisms, such as thread management, synchronizations or data sharing. Nevertheless, although all these skeletons aim to simplify the development of parallel applications, there is no a unified standard [14]. Therefore, users require understanding different libraries, and their capabilities, not only to decide which fits best for their purposes, but also to properly leverage them. Not to mention the migration of applications from one framework to another, which becomes as well an arduous task.

In addition to the Related Work section (given in Chap. 2), this document is organized into two different main parts:

- the first one (Chap. 3) introduces a parallel pattern interface, implementing the initial pattern set defined in D2.1. Different to other object-oriented implementations in the area, the parallel pattern interface, namely **GRPPI**, uses C++ template meta-programming techniques to provide generic interfaces of these patterns without incurring in runtime overheads. An ultimate goal is to accommodate a layer on the top of existing execution environments (e.g. C++11 threads or OpenMP) and pattern-based parallel frameworks (e.g., In-

tel TBB, FastFlow). A section describes the generic and reusable parallel pattern interface **GRPPI**, as the parallel pattern library interfacing threading mechanisms and general parallel programming models of the **RePhrase** project. Eventually a third section conducts 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.

In the second part of this Chapter, we outline how current FastFlow pattern implementation may be fitted into the **GRPPI** "umbrella" as long as with some hints concerning the possibile performance penalties to be used, in particular when using/processing complex data types and/or when more sophisticated parallel patterns are used. The purpose of this third part is to outline what will be produced later on in the project and ideally completed for the last release of the RePhrase pattern software. A preliminary implementation of the **GRPPI** interface on top of a FastFlow backend is also discussed, and the pros and cons related to performance are discussed in detail.

- the second part (Chap. 4) discusses the native interface provided by FastFlow to support the patterns identified in D2.1. This interface is not compliant with the **GRPPI** interface introduced in the first part. Rather, it is the original FastFlow API designed which has been designed such that performance and efficiency are optimized rather than expressive power and/or C++ most recent style programming model compliance.

2. Related work

In the state-of-the-art, multiple works proposing patterns for developing applications to run on modern architectures can be found. Indeed, pattern programming has become one of the best codifying practices in software engineering [13]. The reason is clear: they ease the application structure while achieve a good balance between maintainability and portability applications. In general, these methods started being widely adopted when parallel hardware started arising in desktop computers [18]. In this sense, one of the most common ways to express parallelism is through parallel skeletons or patterns [23]. These patterns can be divided in two main groups: data parallel, e.g., Map, Reduce or MapReduce [3]; and stream parallel patterns, e.g., Pipeline, Farm or StreamFilter [19].

Most of the existing pattern-based frameworks are data-parallel computing oriented. Focusing on implementations targeted to run on multi-core processors, we find solutions such as ArBB [21] and Kanga [17]. ArBB defines a collection of basic data classes and methods to define data-parallel skeletons, which are executed using an abstract machine. The Kanga framework also supports task-parallel skeletons, nevertheless, it lacks of stream-processing patterns. We can also find frameworks that implement data-parallel patterns tailored to accelerators. For example, open-source approaches, like SkePU [12], allow deploying applications to run on both multi-core CPUs and multi-GPU processors. Commercial solutions are also present in the market, such as Thrust [22] and SYCL [16] for CUDA and OpenCL devices, respectively. In both cases, these frameworks use a similar C++ Standard Template Library (STL) to ease the parallelization task. Simultaneously, standardized interfaces are being progressively developed. This is the case of C++ STL algorithms, available in the forthcoming C++17, that start defining parallel versions of already existing STL algorithms [15]. All these frameworks provide high-level interfaces, enabling performance portability between sequential code to multi-core CPUs even GPUs. Although they support a well-established collection of data-parallel patterns, they still lack of stream processing-oriented patterns.

Focusing on libraries that support stream-processing patterns, we encounter a set of well-known frameworks, such as Intel Thread Building Blocks (TBB), FastFlow, and RaftLib. TBB [24] is a C++ parallel framework based on the queue-based parallelism approach. However, it runs best on Intel-based architectures and has no support for accelerators. FastFlow [4, 5] is a skeleton programming

framework, using lock-free communication mechanisms to implement internally its parallel patterns. This approach has support for CUDA and OpenCL. Finally, RaftLib [11] is a C++ template library that aims to fully exploit the stream processing paradigm, supporting dynamic queue optimization, automatic parallelization, and real-time low overhead performance monitoring. In any case, all these parallel frameworks are not usable nor generic enough to be easily leveraged by users when developing parallel applications, making mandatory C++ inheritance techniques to enable stream-processing patterns.

3. Parallel pattern interface

In this chapter, we propose a generic and reusable parallel pattern interface **GrPPI**, implemented in C++, as the parallel pattern library interfacing 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) of the **RePhrase** project. At this stage, **GrPPI** supports the initial set of patterns previously introduced in D2.1 with different programming models, as specified in Tables 3.1 and 3.2. We follow the same classification of patterns listed in deliverable D2.1 for describing the library interfaces proposed. We refer these parallel design patterns "fundamental" as we aim at including in the initial set the most common and thoroughly used known parallel design patterns.

The initial set of **RePhrase** patterns is described both classifying the patterns according to the kind of parallelism captured (stream or data parallelism) and distinguishing the way data to be processed are provided to similarly structured patterns (from external or internal stream sources or from existing data collections, either in-memory or disk based). We also list "sequential" patterns with the purpose of providing the application programmer with patterns suitable to wrap existing sequential ("business logic") code in such a way the code may be used as functional parameter of other patterns (e.g. a stage in pipeline or a worker in a map pattern). In the description of the patterns we also relate data management patterns that can be introduced during the refactoring task.

3.1 Patterns synopsis

According to D2.1 (Sec. 6), patterns can be categorised in three main classes:

- stream parallel (SP), enlisted in Table 3.1.
- data parallel (DP), enlisted in Table 3.2.
- sequential (Seq), they are actually C++ callable objects. In this context, we assume they always run in a single thread.

They can be nested, but not in any order. Arbitrary nesting is not pragmatically useful and may induce useless complexity the definition of parallel behaviour, e.g. in filtering unbound streams of unbound streams. In addition, the experience matured during the design several parallel programming frameworks [2, 3, 6, 9, 25] clearly

Listing 3.1: Pattern structural tree, example.

```
          - Seq {... return optional<>( ...)}
          |
          - Farm - Seq
          |
Pipeline - StreamFilter - Seq
          |
          - Map - Seq
          |
          - Seq {return void}
```

Table 3.1: Stream Parallel patterns vs. Frameworks supported through **GRPPI**.

	Full **GRPPI**						**GRPPI**-like
	Sequential	**OMP**	**TBB**	**Threads**	**FastFlow**	**CUDA Thrust**	**FastFlow Native**
Pipeline	✓	✓	✓	✓	✓	✗	✓
Farm	✓	✓	✓	✓	✓	✓	✓
StreamFilter	✓	✓	✗	✓	✓	✗	✓
StreamAccumulator	✓	✓	✗	✓	✓	✓	✓
StreamIterator	✗	✗	✗	✗	✗	✗	✓

suggests that arbitrary nesting permits any general scalability and performance advantages.

In **GRPPI**, stream parallel patterns can be arbitrarily nested with other stream parallel patterns. They generally behave as *stream filters*, i.e. consume a stream and produce a stream. The Pipeline is always the outer pattern in a stream parallel code. The first stage of the Pipeline generate a stream. It always return a C++ `optional` type, which is used to mark the end-of-stream. The last stage of a Pipeline collapse the stream. It always return `void`. All stages in the middle of a Pipeline are (implicitly) traversed by a stream. Some patterns, as StreamFilter and StreamAccumulator, are designed to be traversed by a stream; they may consume some item of the stream.

Both data parallel or sequential objects can be nested within a stream parallel patterns, but not viceversa. Only sequential object can be nested within a data parallel pattern. Sequential pattern, i.e. C++ callable objects, are always ground objects of a pattern composition, which can be described as a tree, so-called pattern structural tree.

For some stream pattern interfaces (such as Farm) a *helper interface* is also provided. They typically provide the programmer with a compact syntax for the pattern P used in a pipeline with a stream generator (source) and collapser (sink), i.e. Pipeline (source, P, sink).

Table 3.2: Data Parallel patterns vs. Frameworks supported through **GRPPI**.

	Full **GRPPI**						**GRPPI**-like
	Sequential	**OMP**	**TBB**	**Threads**	**FastFlow**	**CUDA Thrust**	**FastFlow Native**
Map	✓	✗	✓	✓	✓	✓	✓
Stencil	✓	✗	✗	✓	✗	✗	✓
Reduce	✓	✗	✗	✓	✗	✓	✓
MapReduce	✓	✗	✗	✗	✗	✓	✓
Divide-and-Conquer	✓	✗	✗	✓	✓	✗	✓

3.2 Stream parallel patterns

In this section we describe the requirements of the *stream parallel* patterns included in the initial **RePhrase** pattern set. Basically, these patterns exploit parallelism in the processing of different items belonging to one or more input data streams. An input data stream is characterized by having a type and by being able to provide items (to be computed) one after the other with a given *interarrival time*.

The stream parallel patterns included in the initial **RePhrase** pattern set include patterns modelling Farm, Pipeline, StreamFilter, and StreamAccumulator computations. Note that the stream generator and collapser patterns, as defined in D2.1, implicitly appear in the lambdas of the **GRPPI** interface, so they have not been defined as individual patterns for the case.

3.2.1 Pipeline

This pattern computes in parallel several stages on a stream item. Each stage processes data produced by the previous stage in the pipe and delivers results to the next stage in the pipe. During the refactoring process, the introduction of this pattern implies the use of a stream generator and collapser patterns. More details about this pattern can be found in Deliverable 2.1, Section 3.2.2.

Interface

The interface for the Pipeline pattern receives the execution model and the functions related to the stages of the Pipeline, in order to receive the items of the input stream, process, and send them to the next stage. As can be seen in Listing 3.2, its C++ interface uses generic programming, i.e., templates. This fact allows users to have a unique interface, so that, it can be reused for any data type. Note as well that is uses variadic templates, so the Pipeline can have an arbitrary number of stages by receiving a collection of functions passed as arguments. As for the first parameter, the execution model received as argument, determines in its structure, specifies how the Pipeline should be executed. For instance, sequential or parallel executions performed underneath by execution models supported by the interface; e.g. for using OpenMP, the structure that should be passed would be

Listing 3.2: Pipeline interface.

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

Listing 3.3: Usage example of the Pipeline pattern.

```
void pipeline_example1( ) {
    int a = 10;
    parallel_execution_thr p{};
    //parallel_execution_ff p{};

    Pipeline( p,
        // Pipeline stage 0
        [&]() {
            a--;
            if (a == 0) optional<int>{};
            else return make_optional(a);
        },

        // Pipeline stage 1
        [&]( int k ) {
            std::string ss;
            ss = "t " + std::to_string( k );
            return std::string( ss );
        },

        // Pipeline stage 2
        [&]( std::string l ) {
            std::cout << l << std::endl;
        }
    );
}
```

`parallel_execution_omp`.

Example

As an example, Listing 3.3 shows an instance of a Pipeline composed of 3 stages that is executed using the OpenMP programming model. These stages, passed as lambda functions, perform the following tasks: *i)* to read the lines of an input file with space-separated values in order to pack them into a vector structure, *ii)* to compute the maximum value in the vector and forward it to the next stage, and *iii)* to print the maximum values onto an output stream. Note that the first and last lambda functions of the Pipeline emulate, respectively, the behavior of the stream generator and collapser patterns, as defined in D2.1. As this Pipeline receives the OpenMP parallel execution model (line 1), its stages are computed in parallel by the 3 worker threads involved in the execution.

Listing 3.4: Farm interface.

```
template <typename ExecMod, typename TaskFunc>
void Farm( ExecMod m, TaskFunc const &farm );
```

Listing 3.5: Farm helper interfaces.

```
template <typename ExecMod, typename InFunc, typename TaskFunc>
void Farm( ExecMod m, InFunc const &in, TaskFunc const &farm );

template <typename ExecMod, typename InFunc, typename TaskFunc, typename
   (cont.)SinkFunc>
void Farm(ExecMod m, InFunc const &in, TaskFunc const & farm , SinkFunc
   (cont.)const &sink)
```

3.2.2 Farm

This pattern computes in parallel the same function $f : \alpha \rightarrow \beta$ over all the items appearing onto an input stream of type α stream delivering the results on the pattern output stream of type β stream. Computations relative to different stream items are completely independent. This pattern is also referred to as *task farm* or *stream map*. During the refactoring process, the introduction of this pattern implies the use of a stream generator and collapser patterns. More details about this pattern can be found in Deliverable 2.1, Section 3.2.1.

Interface

In a similar way, the Farm pattern interface receives the execution mode and one, two or three functions. The very basic interface in Listing 3.4 gets a function and farm it out onto multiple executors; this interface is designed to works on a stream, thus should be nested into another pattern able to generate a stream, e.g. Pipeline. The Farm pattern could be used in insulation by way of its helper interfaces in Listing 3.5, which include a stream source and sink. On the one hand, in case of two functions, the pattern is in charge of *i)* consuming the items from the input stream and *ii)* processing and delivering them individually to the output stream. On the other hand, in case of three, the pattern is in charge of *i)* consuming the items, *ii)* processing, and *iii)* delivering them individually to a specific output stream after transforming each output element.

Note that the `farm` function will be executed in parallel by the different worker threads participating in the execution. Here again, the implementation based on templates makes the interface more flexible and reusable for multiple data types.

Example

Listing 3.6 presents an example of a Farm pattern nested within a Pipeline pattern. In the example, the Farm receive a stream of items, each of them being a `std::vector<int>`. The Farm workers process in parallel different of these items. Once processed, the results items (of type `std::vector<int>` from different workers are gathered and forwarded to the next stage of the Pipeline (i.e. stage 2)

3.2.3 StreamFilter

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 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 interface for the StreamFilter pattern, described in Listing 3.7, receives the execution model, followed by the filter function, which returns a boolean expression. This basic interface is designed to filter a stream, thus it should be used within a pattern producing a stream, e.g. Pipeline. in a boolean expression. `FilterFunc` should return a boolean expression. The stream items are either forwarded to the next stage of the pipeline or dropped if the `FilterFunc` is evaluated `true` or `false`, respectively.

Example

Listing 3.8 shows an example of a StreamFilter pattern nested in a Pipeline in which the filter function (stage 1) discards stream items that are `int` even numbers and pass through odd numbers.

3.2.4 StreamAccumulator

This pattern "sums up" all items appearing on the input stream and delivers results to the output stream. The function used to sum up values ($\oplus$) may be any kind of binary function of type $\oplus : \alpha \times \alpha \rightarrow \alpha$. During the refactoring process, the introduction of this pattern implies the use of a stream generator and collapser patterns. More details about this pattern can be found in Deliverable 2.1, Section 3.2.4.

Listing 3.6: Usage example of the Pipeline +Farm pattern.

```
void pipeline_farm_example() {
    int n=10;
    parallel_execution_ff p{};
    parallel_execution_ff f{};

    Pipeline(p,
        // Pipeline stage 0
        [&]() {
            std::vector<int> v(5);
            for ( int i = 0; i < 5; i++ )
                v[ i ] = i + n;
            if ( n < 0 )
                return optional< std::vector<int> >{};
            n--;
            return make_optional(v);
        },
        // Pipeline stage 1
        Farm(f,
            [&](std::vector<int> v) {
                std::vector<long> acumm( v.size() );
                for(unsigned i = 0; i < acumm.size(); i++ ){
                    acumm[i] = 0;
                    for(auto j : v){
                        acumm[i] += j;
                    }
                }
                return (acumm);
            }
        ),
        // Pipeline stage 2
        [&]( std::vector<long> acc ) {
            double acumm = 0;
            for ( int i = 0; i < acc.size(); i++ )
                acumm += acc[ i ];
            return acumm;
        },
        // Pipeline stage 3
        [&]( double v ) {
            std::cout << v << std::endl;
        }
    );
}
```

Listing 3.7: StreamFilter interfaces.

```
template <typename ExecMod, typename FilterFunc>
void StreamFilter(ExecMod m, FilterFunc const & taskf)
```

Listing 3.8: Usage example of the StreamFilter pattern nested in a Pipeline.

```
void pipeline_filter_example() {
    int a = 10;
    //parallel_execution_thr p{};
    parallel_execution_ff p{};
    parallel_execution_ff f{};

    Pipeline( p,
        // Pipeline stage 0
        [&]() {
            a--;
            if (a == 0) return optional<int>{};
            else return make_optional(a);
        },
        // Pipeline stage 1
        StreamFilter(f, [&]( int k ) {
              if (k%2==0) {
                  std::cout << "Discard␣" << k << "\n";
                  return false;
              } else {
                  std::cout << "Accept␣" << k << "\n";
                  return true;
              }
        }),
        // Pipeline stage 2
        [&]( int k) {
            std::cout << "Sink:␣" << k << std::endl;
        }
    );
}
```

Listing 3.9: StreamAccumulator interface.

```
template <typename ExecMod, typename TaskFunc, typename RedFunc>
void StreamReduce(ExecMod m, TaskFunc const &loc_red, RedFunc const &red)
```

Listing 3.10: StreamAccumulator helper interface.

```
template <typename ExecMod, typename GenFunc, typename TaskFunc, typename
   (cont.)RedFunc>
void StreamReduce(ExecMod m, GenFunc const &in, TaskFunc const &loc_red,
   (cont.)RedFunc const &red)
```

Interface

The StreamAccumulator pattern aims at reducing, using a specific reduction functions, the items appearing on the input stream. Similar to the other stream-oriented interfaces, the StreamAccumulator interface has two variants, the basic interface, designed to be used in a stream shown in Listing 3.9, which receives two lambda expressions to perform its computations: *i)* the `loc_red` function computes the local reductions from the structures received as parameter; and finally *ii)* the `red` function receives the local reductions to produce a global reduced result.

The StreamAccumulator helper interface, shown in Listing 3.10, is designed to be used in insulation. It requires a `in` routine that generate the stream.

Example

As an example, Listing 3.11 shows an instance of a StreamAccumulator pattern summing up the values appearing in the input stream. In this case, the consumer function reads, from the input stream, a list of space-separated integer values, packs them into vectors, and forwards them to the local reduction stage. Then, the reduce function, executed in parallel by the worker threads, picks consecutively these vectors and computes the local sum reduction of its entries. Finally, the local reductions are summed up to produce a total reduced result.

3.3 Data parallel patterns

In this section we describe the requirements of the *data parallel* patterns included in the initial pattern set. Basically these patterns exploit parallelism in the processing of different items or (possibly overlapping) partitions of items belonging to a single "collection" data item. The different data processed in parallel exist at a given point in time, that is there is no need to await them in time as it happens for stream data items.

Listing 3.11: Usage example of the StreamAccumulator pattern.

```
void stream_reduce_example() {
    ifstream is("file.txt");
    int reduce_var = 0;

    parallel_execution_thr p{};

    StreamReduce( p,
        // GenFunc: stream consumer
        [&]() {
            auto r = read_list(is);
            return ( r.size() == 0 ) ? optional<vector<int>>{} :
               (cont.)make_optional(r);
        },
        // TaskFunc: reduce kernel
        [&]( vector<int> v ) {
            int loc_red = 0;
            for( int i = 0; i < v.size(); i++ )
                loc_red += v[i];
            return loc_red;
        },
        // RedFunc: final reduce
        [&]( int loc_red, int reduce_var ) {
            reduce_var += loc_red;
        }, reduce_var
    );
}
```

The data parallel patterns included in the initial pattern set include *map*, *reduce*, *stencil*, *divide and conquer* and *iterative* computations.

3.3.1 Map

This pattern computes a given function $f : \alpha \rightarrow \beta$ over all the data items of an input collection whose elements have type α and produces as output a collection of items of type β hosting the resulting values isomorphic to the input collection. Each item at a generic position i in the output collection come from the computation of the function f onto the data item in the corresponding position of the input collection. This patterns is also known as *parallel for*, *apply-to-all*. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 2.1, Section 3.3.1.

Interface

The interface for the Map pattern, described in Listing 3.12, receives the execution model, followed by the begin and end input iterators, and the begin of the output iterators. Next, it receives as lambda the `map` function and other input iterators than can be eventually needed for the computation of the Map parallel pattern. After-

Listing 3.12: Map interface.

```
template <typename ExecMod, typename InputIt, typename OutputIt, typename
   (cont.)... MoreIn, typename TaskFunc>
inline void Map(ExecMod m, InputIt first, InputIt last, OutputIt firstOut,
   (cont.) TaskFunc const & map, MoreIn ... inputs )
```

Listing 3.13: Usage example of the Map pattern.

```
//Add two vectors
void map_example() {
    std::vector<int> in(1000), in2(1000), out(1000);
    parallel_execution_thr p{8};
    for(int i = 0; i < in.size(); i++){ in[i] = i; in2[i] = i; }

    Map(p, in.begin(), in.end(), out.begin(), [&](int in, int in2){ return
       (cont.) in + in2; }, in2.begin());
}
```

wards, the Map function is executed in parallel by the different working threads, as specified in the execution model. Note that this function should be pure and return the output value for a given element. It is also mandatory that both input and output collections to have the same cardinality, as well as other input data collections that can be received in the variadic template.

Example

Listing 3.13 shows an example of a Map pattern that computes the matrix sum of two input matrices, and writes the result on a third output matrix. Note that the second matrix for the sum operation, is passed as the last argument, following a similar approach to that used in C++ STL algorithms.

3.3.2 Stencil

This pattern computes in parallel the new value of items in an input data collection to be placed at the correspondent position into an isomorph output collection. The computation of the result relative to the item requires as input data some items belonging to the nearer positions of the input collection. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 2.1, Section 3.3.2.

Interface

Similar to the Map pattern, the interface for the Stencil pattern, detailed in Listing 3.14, receives the execution model, the begin and end of the input iterators and

Listing 3.14: Stencil interface.

```
template <typename ExecMod, typename InputIt, typename OutputIt, typename
   (cont.)... MoreIn, typename TaskFunc, typename NFunc>
inline void Stencil(ExecMod m, InputIt first, InputIt last, OutputIt
   (cont.)firstOut, TaskFunc const & stencil, NFunc const & neighborhood,
   (cont.)MoreIn ... inputs )
```

the begin of the output iterator. Next, it receives also the `stencil` function, responsible for computing the results to the output data collection. Finally, it takes the `neighborhood` function, returning a list of neighbours related to a given input iterator position, while the last parameter is intented to receive more input collections for more complex operations.

Example

Listing 3.15 shows an example of a Stencil pattern performing the convolution operation of an input matrix using a given kernel matrix in an image processing use case. As can be seen, the `stencil` lambda expression computes the convolution using the input and kernel matrices, while the `neighborhood` lambda is intented to return a list of values related to a given position of the iterator received as parameter.

3.3.3 Reduce

This pattern "sums up" all the data items of a collection of items of type α using a binary function $\oplus : \alpha \times \alpha \rightarrow \alpha$ that is usually associative and commutative. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 2.1, Section 3.3.3.

Interface

The interface for the Reduce pattern, described in Listing 3.16, receives the execution model, the first and last iterators from the input data collection and a reference to the output data. Next, it receives the `reduction` function, responsible for applying the binary function to reduce the input data.

Example

Listing 3.17 shows an example of a Reduce pattern summming up the entries of a given array. The `reduction` lambda expression accumulates the result inf the out variable and returns it to the user.

Listing 3.15: Usage example of the Stencil pattern.

```
//Convolution
void map_example1() {
    int rowsize = 10;
    std::vector<int> in(100), out(100), kernel(9)
    for(int i=0;i<in.size();i++) in[i] = i;
    for(int i=0;i<kernel.size();i++) kernel[i] = i;

    parallel_execution_thr pe{8};

    Stencil(parallel_execution_omp, in.begin(), in.end(), out.begin(),
        [&](auto it, std::vector<int> &ng){
            if( (it-in.begin()) > rowsize
                && (it-in.begin()) < (in.end()-rowsize-in.begin())
                && ((it-in.begin()) % rowsize) != 0
                && ((it-in.begin()) % rowsize) != (rowsize-1)) {
                auto val = ng[0]*kernel[0] + ng[1]*kernel[1] + ng[2]*
                   (cont.)kernel[2]
                         + ng[3]*kernel[3] + *(it)*kernel[4] + ng[4]*
                            (cont.)kernel[5]
                         + ng[5]*kernel[6] + ng[6]*kernel[7] + ng[7]*
                            (cont.)kernel[8];
                return val;
            }
            return *(it);
        },
        [&](auto it){
            std::vector<int> nn;
            if( (it-in.begin()) > rowsize
                && (it-in.begin()) < (in.end()-rowsize-in.begin())
                && ((it-in.begin()) % rowsize) != 0
                && ((it-in.begin()) % rowsize) != (rowsize-1)) {
                nn.push_back(*(it - 1 - rowsize));
                nn.push_back(*(it - rowsize));
                nn.push_back(*(it - 1 - rowsize));
                nn.push_back(*(it + 1));
                nn.push_back(*(it - 1));
                nn.push_back(*(it - 1 + rowsize));
                nn.push_back(*(it + rowsize));
                nn.push_back(*(it - 1 + rowsize));
            }
            return nn;
        }
    );
}
```

Listing 3.16: Reduce interface.

```
template <typename ExecMod, typename InputIt, typename Output, typename
   (cont.)RedFunc>
inline void Reduce(ExecMod m, InputIt first, InputIt last, Output &
   (cont.)firstOut, RedFunc const & reduce)
```

Listing 3.17: Usage example of the Reduce pattern.

```
//Summ up elements in a vector
Reduce(parallel_execution_omp, in.begin(), in.end(), out,
       [&](int & in, int & out){ out += in; });
```

Listing 3.18: MapReduce interface.

```
template <typename ExecMod, typename InputIt, typename OutputIt, typename
   (cont.)MapFunc, typename RedFunc, typename ... MoreIn>
inline void MapReduce (ExecMod m, InputIt first, InputIt last, OutputIt
   (cont.)firstOut, MapFunc const & map, RedFunc const & reduce, MoreIn ...
   (cont.) inputs)
```

3.3.4 MapReduce

This pattern computes a key value function over all the items of an input connection and eventually delivers a set of unique key value pairs where the value associated to the key is the "sum" of the values output for the same key in the first "map" phase. This pattern is also known as Google mapreduce. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 2.1, Section 3.3.4.

Interface

The interface for the MapReduce pattern, described in Listing 3.18, receives the execution model, the first and last iterators to the input data and the first iterator the output collection. Afterwards, it takes the `map` and `reduce` functions in charge of computing, respectively, the Map and Reduce phases. Optionally, the interface can also receive iterators of other necessary input data.

Example

Listing 3.19 shows an example of a MapReduce pattern computing the matrix-vector multiply. First, the Map phase performs the only the "multiplication" part of dot product, using the rows of the input matrix along with the input vector. The Reduce phase performs the "sum" part of the dot product, so as to generate a final result on the output vector.

3.3.5 Divide-and-Conquer

This pattern computes a problem for which *a)* the solution for some base cases are known and *b)* non-base case problems may be divided into a collection of sub-problems and the solution of the non-base case problems may be computed out of

Listing 3.19: Usage example of the MapReduce pattern.

```
void mapreduce_example1() {
    std::vector<std::vector<int>> mat(10000), v(10000), out(10000);
    for(int i=0;i<mat.size();i++) {
        mat[i] = std::vector<int> (10000);
        for(int j=0;j<mat[i].size();j++){
            mat[i][j] = 1;
        }
    }
    for( int i= 0 ; i< v.size(); i++) {
         v[i] = 2;
    }
    parallel_execution_thr p{8};
    MapReduce(p, mat.begin(), mat.end(), out.begin(),
        [&](auto & in, auto & v){
            std::vector<int> mult(in.size());
            for(auto col = 0; col!= in.size(); col++){
                mult[col] = in[col] * v[col];
            }
            return mult;
        },
        [&](auto & in, int & out){
            out += in;
        },
        v
    );
}
```

Listing 3.20: Divide-and-Conquer interface.

```
template <typename ExecMod, typename Input, typename Output, typename
   (cont.)DivFunc, typename TaskFunc, typename MergeFunc>
void DivideAndConquer(ExecMod m, Input & problem, Output & output, DivFunc
   (cont.) const & divide, TaskFunc const & kernel, MergeFunc const & merge
   (cont.))
```

the solutions of the sub-problems. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 2.1, Section 3.3.5.

Interface

The interface for the Divide-and-Conquer pattern, described in Listing 3.20, receives the execution model, and the references to the problem and output data. Next, it receives three functions: *i)* the `divide` is responsible for splitting the problem in several subproblem, until an indivisibility condition is fulfilled; *ii)* the `kernel` function, in charge of computing the result of an indivisible subproblem; and *iii)* the `merge` function, responsible for merging the partial results in order to obtain a global one.

Example

Listing 3.21 shows an example of a Divide-and-Conquer pattern computing the Fibonacci number passed as argument to the function. To do so, the `divide` function splits the problem in more subproblems, until the Fibonacci number is less than 2. The `kernel` function computes the partial Fibonacci number for a particular subproblem, in an iterative manner. Finally, the `merge` function sums up the partial results to generate a global Fibonacci number, result of that passed initially by argument in the `DivideAndConquer` function call. Note that the `kernel` function computes the Fibonacci number in an iterative way, this is to prevent the creation of as many threads as the amplitude of the binary tree generated during the computation of a Fibonacci number using a parallel `DivideAndConquer` interface. In these cases, the parallel implementations stop dividing the problem when the the number of threads generated in a given point of the computation reaches an upper threshold defined in the parallel execution model.

3.4 Support for parallel patterns on NVidia GPUs

In this section we detail the implementation details for supporting NVidia devices. First of all, the preliminary implementation is based on CUDA Thrust. Since CUDA 7.0, Thrust support C++11, allowing the deployment of lambda function in the GPU. Additionally, CUDA Thrust relies on STL container, specially STL vectors. Thrust defines two main containers: host and device vectors. These containers are fully compatible with standard vectors, so data movement between the host and devices can be implemented is a easy way. Tables 3.1 and 3.2 summarise the implemented parallel patterns on GPU. Stencil pattern is not yet compatible with the current approach due to the lambda implementation restrictions. In contrast to the other patterns, this has to be implemented using a different approach.

The current state of the implementation fully supports multiple GPUs configurations for both stream and data parallel patterns. In case of counting with multiple devices, we deploy the GPU context in a separated thread. Another benefit of this approach is that end-users can configure the GPUs layout by using the environment variable `CUDA_VISIBLE_DEVICES`.

We have defined `parallel_exection_thrust` additional parallel executor. Listing 3.22 shows and example of its usage. In the first line of code, a new parallel executor is created, having as arguments the number of GPUs and the execution policy. The execution policy[1] allows to execute the code over different programming models such as OpenMP and TBB. This approach permits us to even define and create new policies for other non-supported devices like Intel Xeon Phi. In the example, we specify the CUDA-based execution using the `thrust::cuda::par` constant. In the second line of code, we can modify the

[1]More details about Thrust's execution policies at https://thrust.github.io/doc/group__execution__policies.html

Listing 3.21: Usage example of the Divide-and-Conquer pattern.

```
void mapreduce_example1() {
    std::vector<std::vector<int>> mat(100);
    for(int i=0;i<mat.size();i++) {
        mat[i] = std::vector<int> (100);
        for(int j=0;j<mat[i].size();j++){
            mat[i][j] = 1;
        }
    }

    for(int v=0;v<40;v++){
    int find = 1;
    int out = 0;

    DivideAndConquer(parallel_execution_omp, v, out,
        [&](auto & v){
            std::vector< int > subproblem;
            if( v < 2 ) subproblem.push_back(v);
            else {
                subproblem.push_back(v-1);
                subproblem.push_back(v-2);
            }
            return subproblem;
        },
        [&](auto & problem, auto & partial){
            int a = 1, b = 1;
            for( int i = 3; i <= problem; i++ ) {
                int c = a + b;
                a = b;
                b = c;
            }
            partial = b;
        },
        [&](auto & partial, auto & out){
            out += partial;
        }
    );
}
```

Listing 3.22: Usage example of the Thrust-based parallel executor.

```
auto p = parallel_execution_thrust (1, thrust::cuda::par);
cudaGetDeviceCount(&(p.num_gpus));
```

number of available GPUs in the application. This number is always limited by the total number of GPUs available in the system.

In Listing 3.23, we present an example of usage of the Farm pattern running in GPUs. The implemented code is similar to the used in CPU. However, we identify the following restrictions:

- *Containers.* The containers used in lambdas functions are vector-like. These vectors can store any kind of data types, including other vectors.
- *NVidia compiler (NVCC).* Files including these template must be renamed with the extension *.cu* in order to allow the compilation.
- device *statements.* Lambda functions must be marked with the *device* statement. This condition is established by CUDA Thrust.
- *Fine-grained parallelism.* We have to slightly modify the execution stage of the patterns and adapt it. As shown in Listing 3.23, the second lambda iterates over each element of the input list.
- *Capturing variables in lambda functions.* Due a limitation of the current version of Thrust, we can only capture variables by value in lambda functions. The closure operator `[=]` copies needed variables in the GPU main memory, including more sophisticated containers like vectors. In contrast, the closure symbol `[&]` is not yet supported. In any case, memory transfers from/to GPU have to be considered given that they can increase the overall execution time of applications.

3.5 Evaluation

In this section, we perform an experimental evaluation of **GRPPI** 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 by the pattern interface. 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.

Listing 3.23: Usage example of the Farm pattern on GPUs.

```
void farm_example1() {
    int a = 20000;
    auto p = parallel_execution_thrust(1, thrust::cuda::par);
    cudaGetDeviceCount(&(p.num_gpus));

    Farm(parallel_execution_thrust,
        // Farm generator as lambda
        [&]() {
            a--;
            if ( a == 0 ) {
                return optional< vector<float> >();
            }
            else {
                return optional< vector<float> >( vector<float>(1000,a) );
            }
        },
        // Farm kernel as lambda
        [=] __device__ (int float)->float
           {
               return in * a;
           },
        [&](vector<float> v){
           std::cout<<v[0]<<std::endl;
        }


    );
}
```

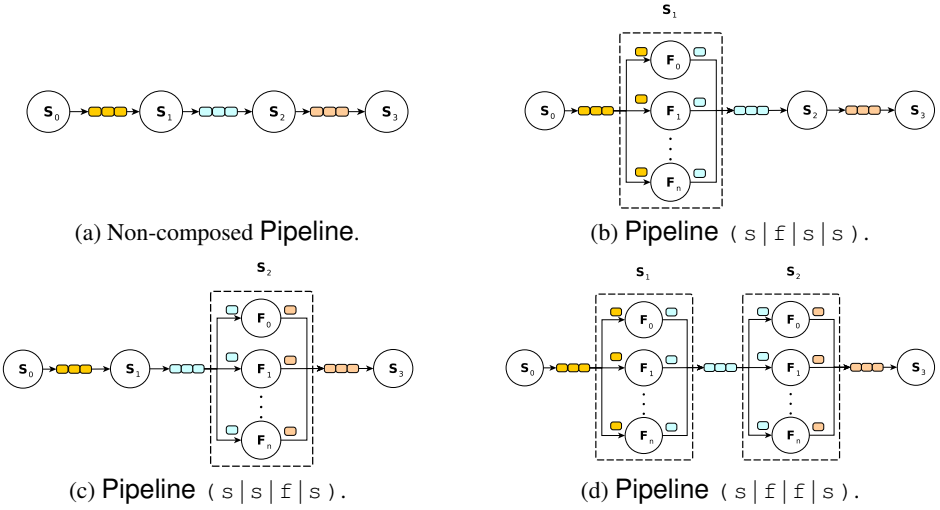
- *Software.* To develop the parallel versions and to implement the proposed interfaces, we leveraged the execution environments C++11 threads and OpenMP, and the pattern-based parallel framework Intel TBB. The C++ compiler used to assemble **GrPPI** is GCC v5.0.

- *Benchmark.* To evaluate the parallel patterns, we used a video stream-processing application composed by two filters, the Gaussian Blur and Sobel operators provided by the OpenCV library, which applied to the frames is capable of detecting edges in the video [5]. Specifically, this application matches the parallel Pipeline pattern, in which the first stage reads the frames from a video file passed as input; the second and third stages apply the Gaussian Blur and Sobel filters, respectively; and the last stage dumps the processed frames to an output video file.

To carry out the experimental evaluation, we first parallelize the aforementioned video application using the above-mentioned execution frameworks and the proposed interface. Afterwards, we compare both performance and lines of code required to implement such parallel versions with respect to the sequential one. Note that for the case of OpenMP, the implementation of the Pipeline pattern is not straightforward: it requires the use of queues to communication items between stages. In our particular case we leveraged a variant of the Michael and Scott lock-free queue in C++ [20]. To further experiment with our interface, we implement different versions of the video application using the execution frameworks and distinct compositions of patterns in its main pipeline. As depicted in Fig. 3.1, (a) we use a non-composed pipeline `( s | s | s | s )`; (b) a pipeline composed of a farm in its second stage `( s | f | s | s )`; (c) a pipeline composed of a farm in its third stage `( s | s | f | s )`; and (d) a pipeline composed of two farms in the second and third stages `( s | f | f | s )`.

3.5.1 Analysis of the usability

In this section we analyze the usability and flexibility of the generic interface developed. To analyze this aspect, we compare the number of lines required to implement the parallel version of the application leveraging the interface, with respect to using directly the parallel execution frameworks. Table 3.3 summarizes the percentage of extra lines introduced into the sequential source code in order to implement such parallel versions using the above-mentioned pattern compositions. As can be seen, implementing more complex pattern compositions via C++ threads or OpenMP leads to larger source codes, while for Intel TBB the number of required extra lines remains constant. Focusing on **GrPPI**, we observe that the effort of parallelizing an application is almost negligible: even the implementation of the most complex composition increases, at most, 4.4 % the total number of lines of code. This behavior is contrary to the C++ threads or OpenMP frameworks, which require roughly twice of lines of code. Additionally, switching **GrPPI** to use a



(a) Non-composed Pipeline. (b) Pipeline (s | f | s | s).

(c) Pipeline (s | s | f | s). (d) Pipeline (s | f | f | s).

Figure 3.1: Pipeline and Farm compositions of the video application.

Table 3.3: Percentage of increase of lines of code w.r.t. the sequential version.

Pipeline composition	% of increase of lines of code			
	C++ Threads	**OpenMP**	**Intel TBB**	**GrPPI**
(s | s | s | s)	+8.8 %	+13.0 %	+25.9 %	+1.8 %
(s | f | s | s)	+59.4 %	+62.6 %	+25.9 %	+3.1 %
(s | s | f | s)	+60.0 %	+63.9 %	+25.9 %	+3.1 %
(s | f | f | s)	+106.9 %	+109.4 %	+25.9 %	+4.4 %

particular execution framework just needs changing a single parameter in the pattern function calls.

3.5.2 Performance analysis of the Pipeline and Farm patterns

Next, we analyze the performance with and without **GrPPI** along with the different execution frameworks and Pipeline compositions, as detailed in Fig. 3.1, for the video application. Concretely, we employ the frames per second (FPS) metric to analyze the behavior of the particular versions using a same input video with diverse resolutions. A first observation is that the Pipeline combined with the Farm pattern for the filtering stages, in comparison to the non-composed Pipeline, improves substantially the FPS for all parallel frameworks. It is also remarkable that for the lowest video resolution, it is only needed to use a Farm pattern in one of the filtering stages in order to attain the maximum performance. However, as the video resolution increases, more complex pattern compositions deliver better FPS rates, given that amount of computation also increases. We also observe that the usage of **GrPPI** does not lead to significant overheads: it is less than 2 %, on average, for all the execution frameworks and compositions. An extra inspection into the plots reveals a corner case for the case of Intel TBB in all Pipeline combinations. This

is due to the TBB implementation intensively relies on dynamic memory allocation primitives to communicate threads, while the **GrPPI**-TBB version employs implicitly C++11 data movement instructions (`std::move`). Finally, we find out that the OpenMP and C++ threads versions with and without **GrPPI** obtain a higher frame rate with respect to the TBB one. This is mainly because we leverage lock-free channels, while TBB internally uses blocking queues.

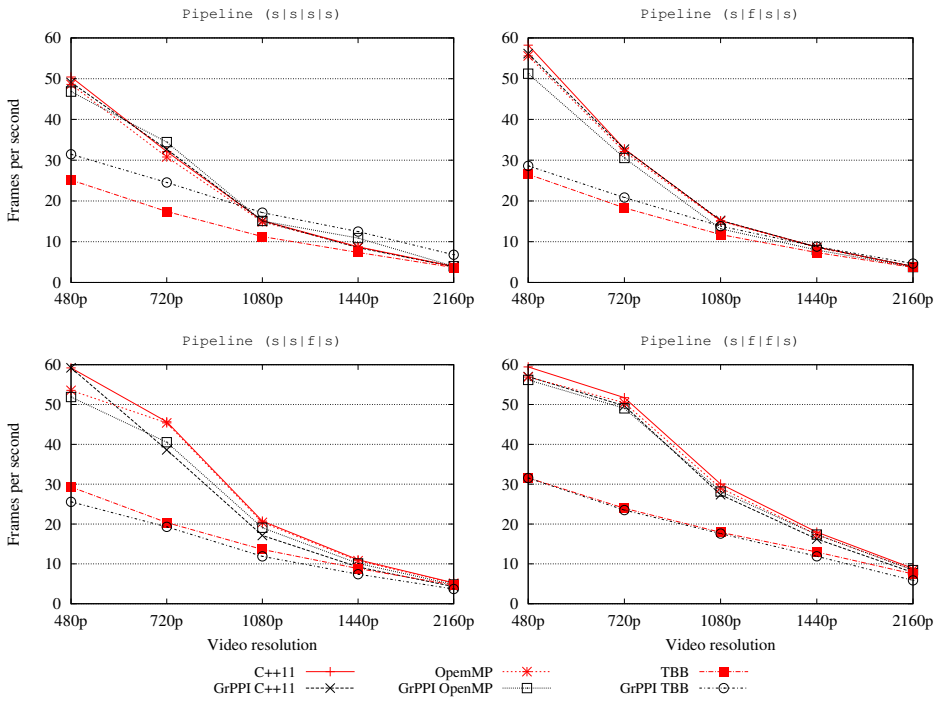
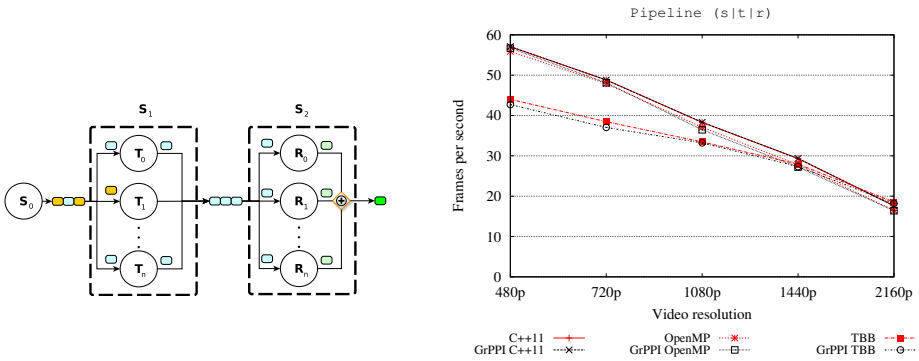


Figure 3.2: FPS w/ and w/o using **GrPPI** along with the different frameworks and Pipeline compositions.

3.5.3 Performance analysis of the StreamFilter and StreamAccumulator patterns

To evaluate the performance of the StreamFilter and StreamAccumulator patterns, we implement a synthetic version of the video application in which we replace both filtering stages to incorporate a filter and a reduce patterns in the stages. Specifically, the filtering stage in charge of discarding video frames whose percentage of black pixels is above a fixed threshold. Finally, the reduce stage computes the number of null pixels in the video frames and displays it to the end user. Fig. 3.3a depicts the Pipeline composition `(s|t|r)` used in this version of the application. Focusing on the FPS attained by the different versions with and without **GrPPI**, as shown in Fig. 3.3b, our main observation is that the interface presented has irrelevant overheads while eases to a large extent the development of parallel

applications.



(a) Pipeline `(s|t|r)`.

(b) FPS w/ and w/o using **GrPPI** along with the different frameworks.

Figure 3.3: Synthetic version of the video application using the StreamFilter and StreamAccumulator patterns.

3.5.4 C++ interface implementation in FastFlow

FastFlow comes as a C++ template library designed as a stack of layers that progressively abstracts out the programming of parallel applications. They are depicted in Fig. 3.4. The goal of the stack is threefold: portability, extensibility, and performance. For this, all the three layers are realised as thin strata of C++ templates that are 1) seamlessly portable; 2) easily extended via subclassing; and 3) statically compiled and cross-optimised with the application.

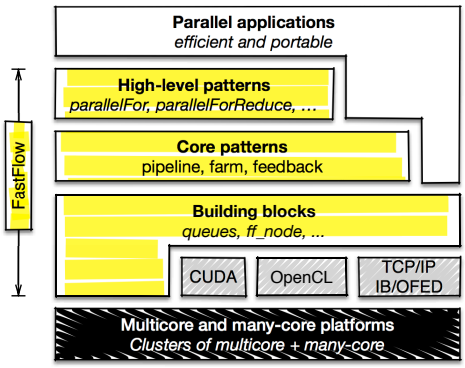


Figure 3.4: FastFlow architecture.

In the hierarchy of abstractions, the **GrPPI** interface shares the very same level of FastFlow "High-level patterns". As a matter of a fact, all the **GrPPI** patterns have a counterpart in FastFlow. Data parallel **GrPPI** patterns can be conveniently implemented directly using these counterparts in a C++ facade design

pattern. Stream parallel **GRPPI** pattern instead is easier to implement by way of the "Core patterns" level of FastFlow. This is because in FastFlow data parallel and stream parallel patterns are implemented at different abstraction levels to support a two-level nesting model: data parallel patterns can be nested within any composition of stream patterns but not viceversa.

Despite similarities, **GRPPI** and FastFlow API represent different design philosophies. They could be categorised in two main groups:

- **Application vs run-time programming**. **GRPPI** is mainly designed for application programming. Whereas FastFlow exhibits a number of advanced programming features that are very useful for building run-time support of other programming frameworks and DSLs, including **GRPPI**. Examples are: support for cyclic networks; support for used-defined scheduling, affinity, and pinning; support for control of input non-determinism in multiple input channels; support for stream item creation and destruction, hybrid blocking-nonblocking behaviour, etc. These features make it possible a very fine control of concurrent behaviour but also makes the API more complex for standard programmers.

- **Shared-memory vs hybrid programming model**. **GRPPI** relies on the shared-memory programming model, whereas FastFlow supports a more general hybrid shared-memory/message-passing programming model. This latter one could be implemented in distribute memory for targeting clusters of multicores [1]. In this model, FastFlow fully support zero-copy data movements. For this purpose, it has a clear edge in terms of performance with respect to other parallel programming frameworks in the cases where data movement is the real application bottleneck, as for example, test reported in Fig. 3.5 and 3.6.

A FastFlow pattern is a network of nodes (implemented as threads) that are instances of the class `ff::ff_node`. A pattern is also a `ff::ff_node`. These patterns can be nested. A node process is a stream of inputs and it is activated on the presence of a stream item in one of its input channel, the node changes state (e.g. running, freezing, terminating) on the reception of special item like an EOS (End Of Stream). A ground node is an actor (implemented as a thread in the shared-memory model). A network of nodes is a network composed by threads where edges are single-producer-single-consumer FIFOs (with either blocking, non blocking, or hybrid behaviour).

In **GRPPI** patterns are composed of C++ callable objects and are callable object themselves. Additionally, these objects can be nested. **GRPPI** can be easily implemented in FastFlow by wrapping all the callable objects within a pattern (also possibly including other patterns) into a FastFlow `ff_node`. The wrapping code is sketched in Listing 3.24. In this way, the eventual concurrent structure of the code will be constructively defined as a network of `ff_nodes`, wrapping all the callable objects, possibly exploited in several replicas. All the *ground* `ff_node`

Listing 3.24: `ff_node` wrapping code

```
template <typename TSin, typename TSout, typename L>
struct PMINode:ff_node {
    L callable;
    PMINode(L const &lf,bool streamitem=true):callable(lf) {};
    inline void * svc(void *t) {
        void * outslot = FF_MALLOC(sizeof(TSout));
        TSout * out = new (outslot) TSout();
        TSin * input_item = (TSin *) t;
        *out = std::move(callable(*input_item));
        input_item->~TSin();
        FF_FREE(input_item);
        return(outslot);
    }
};
```

(i.e. not patterns) will be turned into an actor and implemented as a C++ Thread (in the shared-memory model).

The wrapping shown in Listing 3.24 basically consists in receiving a task, running it by invoking a callable object, and forwarding the result to a consumer node. Due to the copy semantic of **GrPPI**, stream items should be allocated and deallocated. In this wrapping scheme, a specialised FastFlow stream allocator is used to minimise the overhead of the dynamic memory allocation. Memory allocation and copy are the two main source of overhead the FastFlow implementation of **GrPPI** with respect to a native implementation. For the pipeline pattern, this overhead is quantified in Fig. 3.5 and Fig. 3.6. A similar behaviour could be observed for the farm pattern. It should be noticed that in the current version of the **GrPPI** API, this overhead could no be avoided since the move operator in the stream `Item` container is deleted. This aspect could be optimised in future versions of the **GrPPI** interface.

Listing 3.25: pipelinetest

```
#include <iostream>
#include <vector>
#include <fstream>
#include <chrono>
#include <ppi/pipeline.hpp>
using namespace std;
const long wastetime=1000;

struct myitem_t {
    std::string s;
    long n;
    std::vector<long> arr;
    myitem_t(const myitem_t& o): s(o.s),n(o.n), arr(o.arr) { }
    myitem_t():s(""),n(-1) {}
};

void pipeline_example1(long streamlen, long payloadsize, long &sum, long &
   (cont.)items) {
    //parallel_execution_thr p{};
    //parallel_execution_omp p{};
    //sequential_execution p{};
    parallel_execution_ff p{};


    Pipeline( p,
         // Pipeline stage 0
        [&]() {
        streamlen--;
        //for(volatile long j=0;j<wastetime;++j);
        if (streamlen < 0)
            return Item<myitem_t>(); // Preliminary implementation of
               (cont.)option type
        else {
            myitem_t i;
            i.n=streamlen;
            i.s = "";
            i.arr.reserve(payloadsize);
            for (long j=0; j<payloadsize;j++)
                i.arr.push_back(j);
            return Item<myitem_t>(i); // Preliminary implementation of
               (cont.)option type
        }
    },

    // Pipeline stage 1
    [&]( myitem_t k ) {
        std::string s = std::to_string( k.n );
        k.s =  "";
        k.s.reserve(payloadsize*s.length()+2);
        for (int j=0; j<payloadsize;j++)
            k.s += s;
        return k;
    },

    // Pipeline stage 2
    [&](  myitem_t l ) {
        //for(volatile long j=0;j<wastetime;++j);
        ++items;
        sum += l.n;
    }
    );
}
```

Listing 3.26: pipelinetest

```
int main(int argc, char **argv) {
    int payloadsize=20000;
    int streamlen=100000;
    long sum = 0;
    long items = 0;

    if (argc>1) {
        if (argc!=3) {
            std::cerr << "use: " << argv[0] << " streamlen payloadsize\n";
            return -1;
        }
        streamlen=atoi(argv[1]);
        payloadsize=atoi(argv[2]);
    }

    auto start = std::chrono::high_resolution_clock::now();
    pipeline_example1(streamlen, payloadsize,sum, items);
    auto elapsed = std::chrono::high_resolution_clock::now() - start;

    long long microseconds = std::chrono::duration_cast<std::chrono::
       (cont.)microseconds>( elapsed ).count();
    //std::cout << "Execution time : " << microseconds << " us" << std::
       (cont.)endl;
    std::cout << items << "\t" << payloadsize << "\t" << microseconds << "
       (cont.)\t\t" << sum << std::endl;
    return 0;
}
```

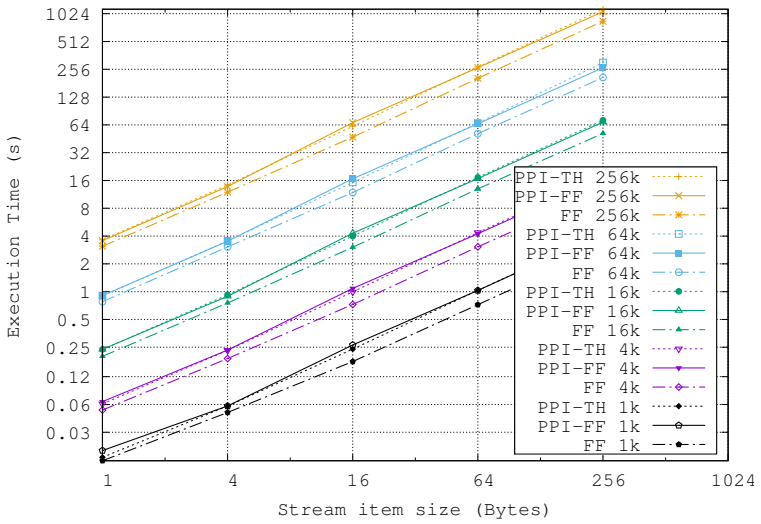


Figure 3.5: Pipeline tests: execution time against different stream item sizes for different stream lengths.

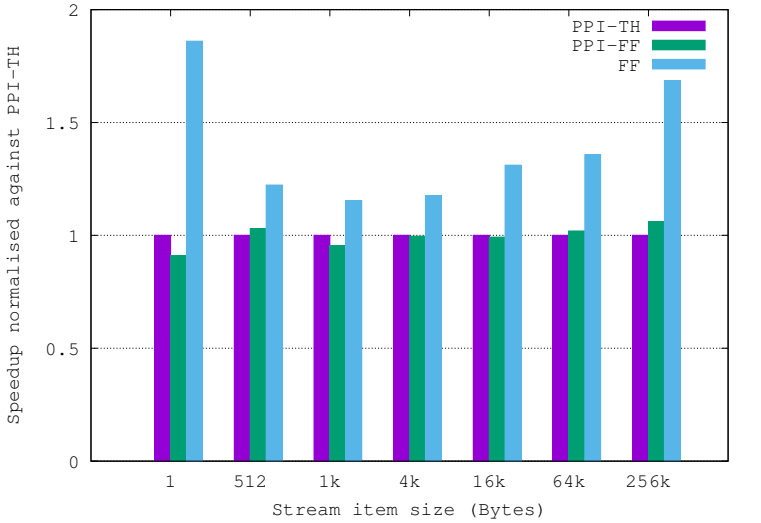


Figure 3.6: Pipeline throughput for different stream item sizes measured as normalised speedup of different frameworks against **GrPPI** Thread.

4. FastFlow native implementation

In this chapter, we will describe the patterns natively provided by the software packages released with this deliverable. We only introduce the API exposed to the application programmer (which is different from the API discussed in Chap. 3) without detailing again the parallel semantics associated to the pattern, which is summarized in the corresponding subsections of Chap. 3 and originally stated in D2.1.

Patterns in FastFlow are declared by instantiating template classes in the FastFlow library. Subsequently, computation of the pattern program is started calling different kind of run methods of the outermost pattern in the pattern composition at hand.

For the sake of simplicity, we only detail how patterns are declared here, as the execution is most of times invoked called the `run_and_wait_end(void)` method of the pattern object. The method starts the computation and awaits its termination in a cooperative way.

4.1 Stream parallel patterns

Before discussing the different stream parallel pattern API, we briefly introduce the simple pattern API used to introduce and remove streams, which is an API that represents the *stream generator* and *stream collapser* patterns discussed in D2.1.

In FastFlow a stream is typically produced (introduced) setting up a `ff_node` (as a sequential wrapper pattern) that outputs stream items through the output verb `void ff_send_out(taskp)` call, where `TypeOut * taskp`. A sample stream generator is provided in Listing 4.1.

A simple wrapper supports stream generation via `std::functions`, which takes two functions accessing a properly initialised state that control whether a new task is to be emitted on the stream and compute the next item of the stream, respectively. The type of the wrapper and typical usage (producing a stream of integers) are shown in Listing 4.2. The wrapper produces an `ff_node` that can be used in any place where a sequential wrapper may be used, e.g. as a pipeline stage or as a farm emitter.

A FastFlow stream collapser may be provided through an `ff_node` wrapper, reading items from its input stream, processing them ,and returning the special

Listing 4.1: Stream generation in FastFlow.

```
template<TypeIn,TypeOut>
  class Source : public ff_node_t<TypeIn,TypeOut> {
    ...
    TypeOut * svc(TypeIn *) {        // called once when the program starts
      while(...) {                 // check end-of-stream case
        Typeout * t = ... ;         // preparing the item to be output
        ff_send_out (t); // this actually outputs an item
      }                            // on the output stream
      return(EOS);                // terminate execution
    }
    ...
  }
```

Listing 4.2: Stream generation in FastFlow using StreamSource.

```
template<class Tstate, class Tout> class StreamSource : public ff_node {
    ...
    StreamSource(Tstate init,
              std::function<bool   (Tstate)>  boolf,
              std::function<Tout * (Tstate*)> valf) { ... }
  }

  StreamSource<int,int>
    source(// initial state value
       16,
       // hasNext() function, more items to be produced?
       [](int x) { return (x==0 ? false : true); },
       // next() function, return new stream item
       [](int * s) { int * n = new int(*s); (*s)--; return(n); }
       );
```

mark `GO_ON` (as shown in Listing 4.3).

4.1.1 Pipeline

Interface

Being one of the patterns included in FastFlow since the very first version of the framework, there are different ways to declare a pipeline pattern in FastFlow. We refer here to the most recent (C++11 compliant) interface.

The pattern is provided through the `ff_Pipe` class, whose interface is described in Listing 4.5. The parameters denote the stages to be added (in order) to the pipeline. An optional, boolean first parameter is used to create the pipeline as an accelerator pattern (see [4]). The pipeline stages, to be passed as parameters to the variadic parameter constructors, **must all subclass** one of the `ff_node`, `ff_node_t` or `ff_node_F` base FastFlow class, which represent a generic FastFlow pattern (`TypeIn*` to `TypeOut*`). This supports pattern compositionality, which is the possibility to use full patterns as pipeline stages.

Listing 4.3: Stream collapsing in FastFlow.

```
template<TypeIn,TypeOut>
  class collapser : public ff_node_t<TypeIn,TypeOut> {
    TypeOut * svc(TypeIn * task) {        // executed foreach item on
       (cont.)input stream
      cout << *task     // process a task (e.g. print it)
           << endl;
      return(GO_ON);                  // continue with the next one
    }
  ...
  };
```

Listing 4.4: Stream collapsing in FastFlow using StreamDrain.

```
  template<class Tstate, class Tin> class StreamDrain : public ff_node {
    ...
    StreamDrain(Tstate init, std::function<void(Tstate *,Tin *)> f) {
      ...
    }

  StreamDrain<int,int>
    drain(0,
      [](int * s, int * t) {
        std::cout << "Task no. " << (*s)++ << " = " << *t << std::endl;
        return; }
      );
```

Example

A sample pipeline pattern declaration is outlined in Listing 4.6. A source stage generates a stream of integers from an initial value (20) down to 1, a stage increases the input integer by 1 and a second stage computes squares it, and eventually a drain stage prints the input numbers. It is worth pointing out how all FastFlow tasks are represented through pointers and this is why the stages compute `TypeOut*(TypeIn*)` functions rather than simpler `TypeOut(TypeIn)` functions.

A more classical example of pipeline (taken from the FastFlow tutorial [7]) is presented in Listing 4.7. In this case, two functions (`F` and `G`) are wrapped adding an additional parameter that is used by the implementation, with the objective of supporting FastFlow internals (the `ff_node *const` parameter) and subsequently used to declare the two `ff_node_F` stages used as pipeline stages.

Listing 4.5: Pipeline FastFlow interface.

```
template<typename IN_t=char,typename OUT_t=IN_t>
class ff_Pipe : ff_pipeline {
  ...
  template<typename... STAGES> ff_Pipe(STAGES &&...stages) { ... }
  template<typename... STAGES> ff_Pipe(bool input_ch,
                                       STAGES &&...stages) { ... }
  ...
  };
```

4.1.2 Farm

Interface

The farm pattern, as the pipeline one, is one of the fundamental patterns present in FastFlow since the very first release of the framework. There are different ways to declare a farm pattern. We outline here the most recent C++11 API. The FastFlow documentation available at the FastFlow web site [8] may be used to read about other APIs supported.

A farm pattern may be declared providing the function computed by the worker and the number of workers to be included to the farm constructor, as shown in Listing 4.8. The `ff_Farm` class being defined as sketched in Listing 4.9.

Example

We consider again the sample code shown in Listing 4.6. Let's assume that for performance reasons we want to parallelize the pipeline stage computing the `sq` function using a farm with 5 workers. The only modification required to introduce the farm leads to the code in Listing 4.10.

4.1.3 StreamFilter

Interface

The pattern is provided through the `ff_SF` class. The interface is described in Listing 4.11. The pattern works on a stream of elements having type `IN_T` and returns a stream of the same type. The first functional argument represents the boolean *predicate* that is used to decide whether or not an input item is to be passed onto the output stream. The second, optional, parameter indicates the number of workers to be used (one by default). The arrival order of the input elements is preserved, i.e. if x_i arrives before x_j and both of them satisfy the predicate, then x_i is sent in output before x_j.

Listing 4.6: Sample Pipeline declaration in FastFlow.

```
int * succ(int * x) { ++(*x); return x; }
int * sq(int * x) { *x *= *x; return x; }

int main(int argc, char * argv[]) {
  ...
  StreamSource<int,int>
    source(16,
       [](int x) { return (x==0 ? false : true); },
       [](int * s) { int * n = new int(*s); (*s)--; return(n); }
       );
  StreamDrain<int,int>
    drain(0,
      [](int * s, int * t) {
        std::cout << "Task no. " << (*s)++ << " = " << *t << std::endl;
        return; }
      );

  FunWrapper<int,int> st1(succ);
  FunWrapper<int,int> st2(sq);
  // or similarly: FunWrapper<int,int> st2([](int * x) { (*x)*=(*x);
     (cont.)return x; });
  ff_Pipe<string,int> pipeline(source,st1,st2,drain);

  pipeline.run_and_wait_end();
  ...
}
```

Example

Listing 4.12 presents an example of application of the pattern. In this case the pattern receives as input a stream of *quotes*, i.e. tuples representing information about the purchase or sale of financial stocks. Each quote is characterized by numerical attributes, such as buying (bid) and selling (ask) volume and price. The filter has to discard quotes that have size less than 20 and respective price less than 120 for the ask or bid fields.

4.1.4 StreamAccumulator

Interface

The Stream-Reduce pattern (also known as Stream Accumulator) is provided by the FastFlow class `ff_SACC`. In the implementation we have distinguished between two notable cases of application:

- the case of an associative and commutative function. In this case, the computation can be performed in parallel by several workers. We consider the following assumption: the function $\oplus$ can be decomposed in two sub-functions $\mathcal{G}$ and $\mathcal{H}$ such that:

$$\oplus(x_1, \ldots, x_K) = \mathcal{H}(\mathcal{G}(x_1, \ldots, x_i), \ldots, \mathcal{G}(x_j, ..., x_K))$$

Listing 4.7: Sample Pipeline declaration in FastFlow.

```
struct fftask_t {
    fftask_t(long r):r(r) {}
    long r;
};
static inline fftask_t* wrapF(fftask_t* in, ff_node*const) {
    long r = F(in->r);
    in->r=r;
    return in;
}
static inline fftask_t* wrapG(fftask_t *in, ff_node*const) {
    long r = G(in->r);
    in->r=r;
    return in;
}

int main(int argc, char * argv[]) {
    ...
    ff_node_F<fftask_t> wrapf(wrapF), wrapg(wrapG);
    ff_Pipe<fftask_t, fftask_t> pipe(wrapf,wrapg);
    ...
}
```

Possibly we could have that $\oplus = \mathcal{G} = \mathcal{H}$ (for example if we have to sum the incoming elements). In this case, each worker applies the computation of a function $\mathcal{G}$ on a subset of the input elements. The function can be applied at the arrival of each incoming elements, updating the partial result accumulated so far. Once K input elements are arrived to the pattern, the partial results are used to compute the final result by using a function $\mathcal{H}$;

- the case of a generic non associative function. In this case the application of the function over K consecutive is performed serially by a single workers. The computation over subsequent groups of K input elements can be performed in parallel by different workers.

To capture these two possibilities, the `ff_SACC` counts with two distinct constructors. The interface for the associative case is shown in Listing 4.13. The pattern receives as input a stream of elements having type `IN_T` and produces elements of type `OUT_T`. An additional type `INTERM_T` can be used if the partial results produced by the workers have a different type with respect to the input one. The programmer has to provide the two functions $\mathcal{G}$ and $\mathcal{H}$:

- the function $\mathcal{G}$ is invoked by a worker each time an input element arrives. It updates an internal state, which will constitute the partial result produced by the worker. Both the input element and the internal state are passed as references;

- the function $\mathcal{H}$ is invoked when the partial results of the workers are available. Its application produces the final result. Also in this case, the partial

Listing 4.8: Sample Farm declaration in FastFlow.

```
struct seq: ff_node_t<ff_task> {
    ff_task *svc(ff_task *t) {
      // process task and possibly modify it
      ...
      return t;
    }
};

int main(int argc, char * argv[]) {
  ...
  ff_Farm farm<task_in_t,task_out_t> farm(f,10);
  ...
}
```

Listing 4.9: `ff_Farm` class in FastFlow.

```
template<typename IN_t=char, typename OUT_t=IN_t>
class ff_Farm: public ff_farm<> {
  ...
  ff_Farm(std::vector<std::unique_ptr<ff_node> > &&W,
          std::unique_ptr<ff_node> E  =std::unique_ptr<ff_node>(nullptr),
          std::unique_ptr<ff_node> C  =std::unique_ptr<ff_node>(nullptr),
          bool input_ch=false) { ... }
  ...
  template <typename FUNC_t>
    explicit ff_Farm(FUNC_t F, ssize_t nw, bool input_ch=false) {
      ... }
  ...
 }
```

results (stored in a `vector`) and the final result to produce are passed by reference;

- the number of items `k` over which compute and the desired number of workers to be used (`nw`, equal to one by default).

Listing 4.14 shows the interface for the non associative case. Also in this case the pattern receives elements of type `IN_T` and produces elements of type `OUT_T`. The functional parameters to be provided are:

- the function $\mathcal{F}$ to be computed over K consecutive elements. The input elements over which compute the function (stored in a `vector`) and the variable in which store the final result are passed by reference;

- the number of items `k` over which compute and the desired number of workers to be used (`nw`, equal to one by default).

In both cases (associative and non associative), it is required that all the used types have a default constructor. Results are produced ordered, i.e. the result of the

Listing 4.10: Farm sample code in FastFlow.

```
int main(int argc, char * argv[]) {

  ...
  FunWrapper<int,int> st1(succ);
  // commented: FunWrapper<int,int> st2(sq);
  // substituted by :
  // wrap the sq into a worker function
  auto f = workerWrap<int,int>(sq);
  // create the farm
  ff_Farm<int,int> st2(f,5);
  // end modification

  ff_Pipe<string,int> pipeline(source,st1,st2,drain);

  pipeline.run_and_wait_end();

  return(0);
}
```

Listing 4.11: Stream Filter FastFlow interface.

```
template<typename IN_T>
ff_SF(const std::function<bool(const IN_T &)> predicate, unsigned int nw
   (cont.)=1)
```

computation over the input elements $x_i, \ldots, x_{i+K}$ is sent before the result of the computation over $x_{i+K+1}, \ldots, x_{i+2K}$.

Example

Listing 4.15 shows an example of application of the `ff_SACC` pattern in the case of an associative function. In the example, the pattern receives and produces a stream of `long` elements. Both function $\mathcal{G}$ and $\mathcal{H}$ are essentially sum operations.

4.1.5 StreamIterator

Interface

The StreamIterator pattern is implemented by the `ff_SI` class, whose interface is described in Listing 4.16. The pattern works on a stream of type `IN_T` and iterates the computation of another pattern. The parameters that must be provided are:

- the *nested* pattern to iterate. In FastFlow every pattern is a subclass of `ff_node` and therefore 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;

Listing 4.12: Filter usage example.

```
ff_SF<quote_t> stream_filter( [](const quote_t &t){
        if(t.ask_size<20 && t.ask_price<120 )
            return false;
        if(t.bid_size<20 && t.bid_price<120 )
            return false;
        return true;
    },nworkers);
```

Listing 4.13: `ff_SACC` FastFlow interface for associative function.

```
template<typename IN_T, typename OUT_T=IN_T, typename INTERM_T=IN_T>
ff_SACC(const std::function<void (const IN_T &,INTERM_T&)> g, const std::
   (cont.)function<void (const std::vector<INTERM_T> & ,OUT_T&)> h ,
   (cont.)unsigned int k, unsigned int nw=1)
```

Listing 4.14: `ff_SACC` FastFlow interface for non associative function.

```
template<typename IN_T, typename OUT_T=IN_T>
ff_SACC( const std::function<void (const std::vector<IN_T> &, OUT_T &)> f,
   (cont.)unsigned int k,unsigned int nw=1)
```

- a `predicate` stating whether the result of the nested pattern must be flown again to the nested pattern input stream;
- 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` 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 4.17 shows a simple example of use of the `ff_SI` 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` pattern impose that the stream elements have to be flown again if they are multiple of 2 or 3. In addition, in the pattern declaration we

Listing 4.15: `ff_SACC` usage example.

```
ff_SACC<long> sacc([](const long& input_t, long &internal_state){
        internal_state+=input_t;
    }, [] (const vector<long > &partial_results,long &ret){
        for(int i=0;i<partial_results.size();i++)
        {
            ret+=(partial_results[i]);
        }
    },k,nworkers);
```

Listing 4.16: `ff_SI` FastFlow interface.

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

indicate to give more priority to re-flown elements with respect to the input stream elements.

4.2 Data parallel patterns

4.2.1 Map

Interface

The map pattern in FastFlow is provided through `ParallelFor` pattern. Different variant exists, modelling independent iteration loops as well as loops that accumulate results in a "reduction" variable, similarly to what provided in OpenMP parallel for pragmas. The implementation of the parallel for uses a farm pattern inside, exploiting all the peculiarities and possibilities offered.

The interface provided is outline in Listing 4.18. The parallel for declarations allows to specify that spinwait based barrier have to be used. Once declare, the `parallel_for` method starts the computation of the pattern. The iteration body is provided through the function parameter, after specifying iteration variable bounds and (possibily) increase/decrease steps.

A `ParallelForReduce` patter is provided as well, supporting the reduction of variable across iterations. The class outline is given in Listing 4.19. Two "reduction" functions are used as parameters; the first one, reduces locally to a single iteration space partition, taking as argument the index of the item begin processed, the second one reduces the reduced values from the differnet partitions, taking as arguments the total reduction variable and the partition computed value.

Listing 4.17: `ff_SI` 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<long> si(nested_pattern,[](const long &t){
            return((t%2)==0 || (t%3)==0);
        }, ff_SI<long>::SIPriority::NESTED_PATTERN);
```

Listing 4.18: `ParallelFor` FastFlow interface.

```
  class ParallelFor: public ff_forall_farm<forallreduce_W<int> > {
    ...
    explicit ParallelFor(const long maxnw=FF_AUTO, bool
                         spinwait=false, bool spinbarrier=false) {
      ...
    }
    ...
    template <typename Function>
    inline void parallel_for(long first, long last, const Function& f,
                             const long nw=FF_AUTO) { ... }
    template <typename Function>
    inline void parallel_for(long first, long last, const Function& f,
                             const long nw=FF_AUTO) { ... }
    template <typename Function>
    inline void parallel_for(long first, long last, long step, long grain,
                             const Function& f, const long nw=FF_AUTO)
                             { ... }
  }
```

Example

Sample code outlining the usage of a parallel for is given in Listing 4.20. In this case a parallel for is declared mapping the `A[j]=j+k;` computation over all the items in array `A`.

4.2.2 Stencil & StencilReduce

Interface

The stencil patterns in FastFlow provide the possibility to instantiate a stencil computation on 2D array data structure [3]. The template class is outlined in Listing 4.21. The stencil pattern is declared passing all the needed parameters, then the computation of the stencil may be started using the `compute` methods.

Listing 4.19: Parallel for reduce in FastFlow.

```
template<typename T>
class ParallelForReduce: public ff_forall_farm<forallreduce_W<T> > {
  ...
  explicit ParallelForReduce(const long maxnw=FF_AUTO, bool
                             spinwait=false, bool spinbarrier=false)
    { ... }

  template <typename Function, typename FReduction>
    inline void parallel_reduce(T& var, const T& identity,
                                long first, long last,
                                const Function& partialreduce_body, const
                                   (cont.)FReduction& finalreduce_body,
                                const long nw=FF_AUTO) { ... }
}
```

Listing 4.20: Sample parallel for code in FastFlow.

```
long *A = new long[size];
...
ParallelForReduce<long> pfr(nworkers, true);
long sum = 0;
for(int k=0;k<ntimes; ++k) {
           auto loop1 = [&A,k](const long j) { A[j]=j+k; };
           auto loop2 = [&A](const long i, long& sum) { sum += A[i];};
           auto Fsum = [](long& v, const long elem) { v += elem; };

           pfr.parallel_for(0L,size,1L,chunk,loop1,nworkers);
           ...
           pfr.parallel_reduce(sum,0L,0L,size,1L,chunk,loop2,Fsum,
              (cont.)nworkers);
           ...
}
```

Listing 4.21: Stencil FastFlow interface.

```
// functions used to compute the stencil value on the i,j point have type
typedef std::function<T(long i, long j,                // point coords
                        T *in,                         // input data
                           (cont.)structure
                        const size_t X, const size_t Y, // array
                           (cont.)dimensions
                        T& reduceVar                   // red. variable
                        )> reduce_F_t;

template<typename T>
class stencil2D: public ff_node {
  ...
  // constructors
  // params: input and output data structures, dimensions, par degree,
     (cont.)stencil radius,
  //         implementation with gosth cell flag, partitioning chuncksize
  stencil2D(T *Min, T *Mout, const size_t Xsize, const size_t Ysize, const
     (cont.) size_t Youtsize,
            int nw, int Yradius=1,int Xradius=1,bool ghostcells=false,
            const size_t chunksize=DEFAULT_STENCIL_CHUNKSIZE) { ... }
  stencil2D(int nw, int Yradius=1, int Xradius=1,bool ghostcells=false,
            const size_t chunksize=DEFAULT_STENCIL_CHUNKSIZE) { ... }

  ...
  // compute methods: the only needed parameter is the stencil compute fun
  void computeFunc(reduce_F_t F,
                   size_t xstart=0, size_t xstop=0, size_t xstep=1,
                   size_t ystart=0, size_t ystop=0, size_t ystep=1,
                   size_t zstart=0, size_t zstop=0, size_t zstep=1)
                   { ... }
 void computeFuncAll(reduce2_F_t F,
                     size_t xstart=0, size_t xstop=0, size_t xstep=1,
                     size_t ystart=0, size_t ystop=0, size_t ystep=1,
                     size_t zstart=0, size_t zstop=0, size_t zstep=1)
                     { ... }
}
```

Listing 4.22: Stencil GPU FastFlow interface.

```
template<typename T, typename TOCL = T, typename accelerator_t =
   (cont.)ff_oclAccelerator<T, TOCL> >
class ff_stencilReduceLoopOCL_1D: public ff_oclNode_t<T> {
  ...
  ff_stencilReduceLoopOCL_1D(const std::string &mapf,
                             const std::string &reducef = std::string(""),
                             const Tout &identityVal = Tout(),
                             ff_oclallocator *allocator = nullptr,
                             const size_t NACCELERATORS = 1,
                             const int width = 1)
                             { ... }
```

It is worth pointing out that, as for the parallel for pattern, stencil patterns are provided within FastFlow targeting GPUs rather than CPU cores. Two pattern classes are provided `ff_stencilReduceOCL` and `ff_stencilReduceCUDA` that support the execution of stencil operations (as well as maps and reduces operations as subclasses) targeting GPUs through either OpenCL or CUDA. The only relevant difference is that the functions to be used to compute the stencil, the map or the reduce operations must be provided using macros whose parameters host all the information necessary to build either the host or the device version of these kernel function. As an example, a simple sum reduction function should be given with a `FFREDUCEFUNC(name, T, param1, param2, code)` macro such as

```
FFREDUCEFUNC(plus,double,x,y, return(x+y);)
```

The interface of the classes targeting the GPUs through OpenCL is outlined in Listing 4.22. The main feature to point out here is that targeting GPUs, which are accelerators with limitations in the access modes supported to the CPU main memory, explicit task definition code is needed to specify what has to be moved to and from the accelerator memory (see sample code in Listing 4.24).

Example

Assume we want to compute the following sequential stencil code in parallel

```
while(k<=maxit && error > tot) {
  /* copy new solution into old matrix */
  for(int j=0;j<m;j++)
    for(int i=0;i<n;++i) uold[i+m*j]=u[i+m*j];
  /* computes the stencil and residual */
  for(int j=1;j<(m-1);++j)
     for(int i=1;i<(n-1);++i) {
        resid = compute_resid(f,i,j, uold,ax,ay);
        /* updates solution */
        u[i + m*j] = uold[i + m*j] - omega * resid;
        /* accumulates residual error */
        error =error + resid*resid;
     }
  error = sqrt(error) / (n*m);  k++;
```

Listing 4.23: Sample stencil usage in FastFlow.

```
// instantiate stencil pattern
stencil2D<double> stencil(u,uold,m,n,n,NUMTHREADS,1,1,false);

stencil->initInFunc(initU);
stencil->initOutFunc(initUold);

stencil->computeFunc(stencilF, 1,m-1,1, 1,n-1,1);
stencil->reduceFunc(condF, maxit, reduceOp, 0.0);

stencil->run_and_wait_end();
```

```
}
```

It may be implemented using a simple stencil skeleton instantiation such as the one in Listing 4.23 where:

- *initU* initializes the single cell of the u matrix. The function is called in a parallel loop in order to execute the initialization phase in parallel;
- *initUold* initializes the single cell of the $uold$ matrix as in the previous case;
- *computeFunc* executes the stencil for each pair (i, j) updating the u matrix and reading values from the $uold$ matrix;
- *reduceFunc* reduction function used to evaluate the error and for terminating the computation;
- *run_and_wait_end* starts the stencil computation and wait for termination.

Sample code using a stencil pattern (its mapreduce subclass, actually, assuming the stencil is limited to the current item only) is shown in Listing 4.24.

4.2.3 MapReduce

Interface

The MapReduce pattern listed in D2.1 model the Google Map Reduce parallel pattern. A stream of items is processed to generate key, value pairs, then the pairs with the same key are processed to "sum up" all the values. Eventually a set of key, value pairs is output as the map reduce result.

Two versions of the Google map reduce are provided within FastFlow:

- the first version accepts as input a stream of items of type α, uses each item to produce an internal stream of items of type β and eventually processes the β items to obtain $\langle \gamma, \delta \rangle$ pairs through a $f : \beta \rightarrow \langle \gamma, \delta \rangle$ function. The values corresponding the same key of type γ are eventually summed up through a $\oplus : \gamma \rightarrow \gamma \rightarrow \gamma$ function.

Listing 4.24: Sample data parallel computation targeting GPU through OpenCL in FastFlow.

```
FF_OCL_MAP_ELEMFUNC_1D_ENV(mapf, float, a, float, b,
                           return (a * b);
);

FF_OCL_REDUCE_COMBINATOR(reducef, float, x, y,
                         return (x+y);
);
#endif

struct oclTask: public baseOCLTask<oclTask, float, float> {
    oclTask():M(NULL),M2(NULL), Mout(NULL),result(0.0), size(0) {}
    oclTask(float *M, float *M2, size_t size):M(M),M2(M2),Mout(NULL),
       (cont.)result(0.0),size(size) {
        Mout = new float[size];
        assert(Mout);
    }
    ~oclTask() { if (Mout) delete [] Mout; }

    void setTask(oclTask *t) {
       setInPtr(t->M, t->size);
       setEnvPtr(t->M2, t->size);
       setOutPtr(t->Mout, t->size);
       setReduceVar(&(t->result));
     }

    float combinator(float const &x, float const &y) {return x+y;}

    float *M, *M2;
    float  *Mout, result;
    const size_t  size;
};

int main(int argc, char * argv[]) {
    size_t size = 1024;
    if (argc>1) size     =atol(argv[1]);
    printf("arraysize = %ld\n", size);

    float *M = new float[size];
    float *M2 = new float[size];

    for(size_t i=0;i<size;++i) {
        M[i] = 2.0f;
        M2[i] = 5.0f;
    }

    oclTask oclt(M, M2, size);

    ff_mapReduceOCL_1D<oclTask> oclMR(oclt, mapf, reducef, 0, nullptr,
       (cont.)NACC);
    ...
}
```

Listing 4.25: MapReduce FastFlow interface.

```
template<class MapReduceInputType,class GenState,class MapInputType,
   (cont.)class KeyType, class ValueType>
class GoogleMapReduce

GoogleMapReduce(   // constructs a version with internal stream
   (cont.)generation
   GoogleMapReduceStreamSource<GenState,MapReduceInputType,MapInputType>
      (cont.) * source,
   std::function<KeyValueP<KeyType,ValueType>*(MapInputType*)> f,
   std::function<ValueType(ValueType,ValueType)> oplus,
   int nw1,
   int nw2
)

GoogleMapReduce(   // constructs a version working on the external
   (cont.)stream
   std::function<KeyValueP<KeyType,ValueType>*(MapInputType*)> f,
   std::function<ValueType(ValueType,ValueType)> oplus,
   int nw1,
   int nw2
)
```

- the second version is a simplified version defined as in D2.1. It accepts an input stream of items of type α, turns it into a stream of β, γ pairs through a function $f : \alpha \rightarrow \langle \beta, \gamma \rangle$. The items with the same key are then summed up through function $\oplus : \gamma \rightarrow \gamma \rightarrow \gamma$ and the result is delivered on the output stream.

The interfaces for the two implementations are given in Listing 4.25.

Example

Listing 4.26 shows the code needed to implement *word count* using the MapReduce FastFlow patttern implementation.

4.2.4 Divide-and-Conquer

Interface

The Divide-and-Conquer pattern is provided by FastFlow through the `ff_DC` class. Its interface is detailed in Listing 4.27.

The pattern receives in input data having type `OperandType` and returns a result of type `ResultType`. The programmer has to specify:

- the `divide` function to split the problem in sub-problems. Sub-problems must be inserted into the `vector` passed by reference;
- a `combine` function that builds the result of a problem starting from the solution of its sub-problems;

Listing 4.26: Word count with FastFlow MapReduce.

```
#include "googlemapreduce.hpp"
...
int main(int argc, char * argv[]) {
  ...
  // function parameters for the MapReduce StreamGenerator
  auto init_ifl = [](string *fn) {
    ifstream * ifl = new ifstream {};   // declare the state
    ifl->open(*fn);                     // open the file (ignore errors,
       (cont.)TODO)
    return ifl;                         // return the state pointer
  };
  auto fin_ifl  = [](ifstream *ifl) {
    ifl->close();                       // close file
    delete ifl;
    return;
  };
  auto hasmore  = [](ifstream *ifl) {
    return(! (ifl->eof()));             // if the file is ended, there are
       (cont.) no more strings to output
  };
  auto nextitem = [](ifstream *ifl) {   // called after checking with
     (cont.)hasmore() that there is something else to read
    string * s = new string {};         // allocate string
    (*ifl) >> (*s);
    return s;
  };
  // declare the mapreduce internal stream generator
  GoogleMapReduceStreamSource<ifstream,string,string> source1(init_ifl,
     (cont.)fin_ifl, hasmore, nextitem);
  auto f = [](string * s) {             // this is F
    KeyValueP<string, long> *kv = new KeyValueP<string,long > {*s, 1};
    delete s;
    return kv;
  };
  auto oplus = [](long a, long b) { return (a + b); }; // this is oplus
  SourceFn src { streamno, fn }; // provides a stream of filenames (as
     (cont.)string)
  GoogleMapReduce<string, ifstream, string, string, long> mapred(&source1,
     (cont.) f, oplus, nw1, nw2);
  // main parallel program
  ff::ff_pipeline Main { };
  Main.add_stage(&src);
  Main.add_stage(&mapred);
  Main.add_stage(new DrainM { });  // DrainM prints the input std::map<
     (cont.)string,long> items
  // execution
  Main.run_and_wait_end();
  return(0);
}
```

Listing 4.27: Divide-and-Conquer FastFlow interface.

```
template <typename OperandType, typename ResultType>
ff_DC(const std::function<void (const OperandType&, std::vector<
   (cont.)OperandType> &)>& divide,
    const std::function<void(std::vector<ResultType>&,ResultType&)>&
       (cont.)combine,
    const std::function<void(const OperandType&, ResultType&)>& base_case,
    const std::function<bool(const OperandType&)>& cond,
    const OperandType& op, ResultType& res, int numw)
```

- a `base_case` solution for the base case problem, i.e. a problem that can be solved directly;
- a `condition` to check if the input problem is a base case problem;
- the reference to the starting problem (`op`) and the final result (`res`) that must be computed;
- the number of workers to use.

All the functional parameters must be provided as `std::function`, i.e. they can be any callable C++ object such as function pointers or lambda expressions. The `ff_DC` class extends the `ff_node_t` class and therefore can be composed with other FastFlow patterns. Once that an `ff_DC` object is constructed, the computation can be started by invoking the `run_and_wait_end()` method.

Example

Listing shows and example of the `ff_DC` pattern, computing the n^{th} number of Fibonacci. Problems are subdivided until we reach the base case $n \leq 2$.

Listing 4.28: Divide-and-Conquer usage example.

```
ff_DC<unsigned int, unsigned int> dac(
                [](const unsigned int &op,std::vector<unsigned int> &
                   (cont.)subops){
                        subops.push_back(op-1);
                        subops.push_back(op-2);
                    },
                [](vector<unsigned int>& res, unsigned int &ret){
                        ret=res[0]+res[1];
                    },
                [](const unsigned int &op, unsigned int &res){
                        res=1;
                    },
                [](const unsigned int &op){
                        return (op<=2);
                    },
                n,
                res,
                nworkers
    );
```

5. Conclusions and future works

In this deliverable, we have reviewed the state-of-the-art about and have presented a general and reusable parallel pattern interface, namely **GRPPI**, which leverages modern C++ features, metaprogramming concepts, and template-based programming to act as switch between parallel programming models. Its compact design facilitates the development of parallel application, hiding away the complexity behind the use of concurrency mechanisms.

As observed throughout the evaluation with a parallel stream-oriented video application, the performance attained by each combination of parallel patterns using diverse parallel frameworks directly, with respect to using **GRPPI** is almost the same. We prove as well that our approach does not lead to considerable overheads while permits to easily parallelize application by adding, on average, 4.4 % of lines of codes. With **GRPPI**, we advocate for a usable, simple, generic, and high-level stream an data parallel pattern interface, allowing users to easily parallelize applications without requiring a deep understand of existing parallel programming frameworks or third-party interfaces.

We also outlined the interface relative to the patterns discussed in D2.1 proposed by FastFlow, which is different from the one implemented in **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, some of the stream parallel FastFlow patterns have been wrapped in **GRPPI** to test a) the feasibility of enxapsulation of a FastFlow im-

plementation 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 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 extend **GrPPI** for supporting more stream and data parallel patterns, as well as more specific parameters and API extensions for the existing patterns to support specific pattern use cases, in particular those arising from the WP6 Use cases, kernels and applications. Furthermore we plan to include in the final version of the WP2 software within RePhrase more execution environments, in particular the full FastFlow execution environment.

Bibliography

[1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in fastflow. In *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing*, volume 7640 of *LNCS*, pages 47–56. Springer, 2013.

[2] Marco Aldinucci and Marco Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, October 2007.

[3] Marco Aldinucci, Marco Danelutto, Maurizio Drocco, Peter Kilpatrick, Claudia Misale, Guilherme Peretti Pezzi, and Massimo Torquati. A parallel pattern for iterative stencil + reduce. *Journal of Supercomputing*, 2016. To appear.

[4] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating Code on Multi-cores with Fastflow. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 170–181, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] 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.

[6] Marco Aldinucci, Marco Danelutto, and Paolo Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.

[7] Marco Aldinucci, Marco Danelutto, and Massimo Torquati. Fastflow tutorial. Technical Report TR-12-04, Università di Pisa, Dipartimento di Informatica, Italy, March 2012.

[8] Marco Aldinucci and Massimo Torquati. *FastFlow website*, 2016 (last accessed). http://mc-fastflow.sourceforge.net/.

[9] Marco Aldinucci, Massimo Torquati, and Massimiliano Meneghin. FastFlow: Efficient parallel streaming applications on multi-core. Technical Report

TR-09-12, Università di Pisa, Dipartimento di Informatica, Italy, September 2009.

[10] 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.

[11] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 96–105, New York, NY, USA, 2015. ACM.

[12] 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.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[15] Jared Hoberock. N4507: Programming Languages - Technical Specification for C++ Extensions for Parallelism. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf, May 2015.

[16] Khronos OpenCL Working Group. SYCL: C++ Single-source Heterogeneous Programming for OpenCL. https://www.khronos.org/sycl. [Last access May 2015].

[17] D. Kist, B. Pinto, R. Bazo, A. R. D. Bois, and G. G. H. Cavalheiro. Kanga: A skeleton-based generic interface for parallel programming. In *2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, pages 68–72, Oct 2015.

[18] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[19] 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.

[20] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[21] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 224–235, April 2011.

[22] NVIDIA Corporation. Thrust. https://thrust.github.io/.

[23] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, London, UK, UK, 2003.

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

[25] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.