Project no. 644235

REPHRASE

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

Final Report on Patterns and Relationship with General Design Patterns
D2.7

Due date of deliverable: M30–Sept. 30, 2017

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

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

Version 1.0

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	20/09/17	Gabriele Mencagli	UNIPI	First version of the document
2	29/09/17	Marco Danelutto	UNIPI	Added executive summary, Introduction and Conclusions
3	30/09/17	Marco Danelutto & Gabriele Mencagli	UNIPI	Deliveable completed
4	2/10/17	Chris Brown	USTAN	Proofread

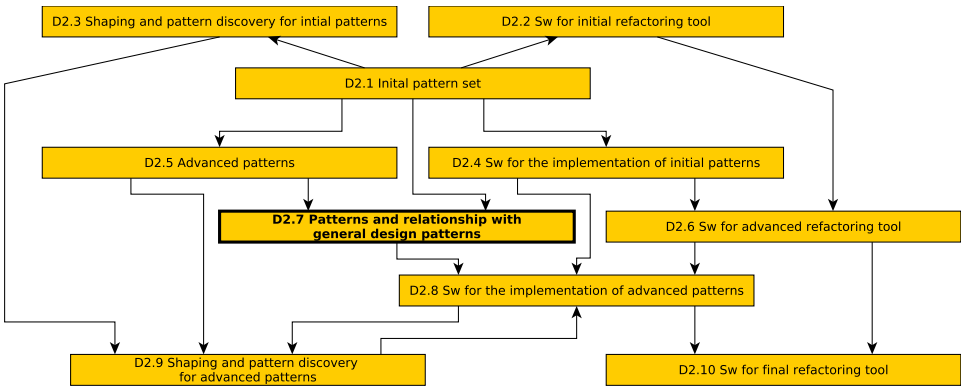
Executive Summary

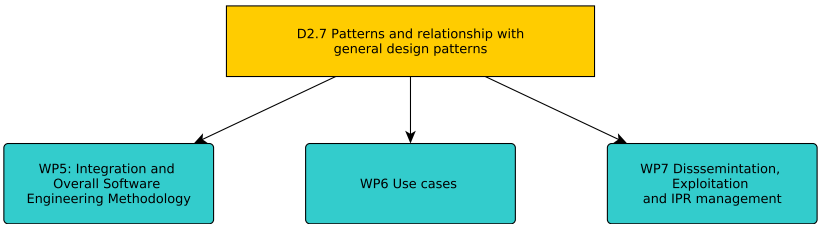
The deliverable reports on the **RePhrase** experience related to the usage of the parallel design patterns with respect to the assessed results and experiences concerning parallel design patterns in general. The first part of the paper introduces and discusses the general design pattern concepts. Then we summarize the pattern design experience through **RePhrase** and eventually we assess our main achievements w.r.t. the most general parallel pattern concepts and experiences, both related to usability and to specific aspects as developed during the project so far.

This deliverable is one of the last deliverables from WP2, being followed by the deliverables concerning the final, tuned release of the pattern libraries used as target by the refactoring tools of **RePhrase** and as parallel building block by the use case developers within **RePhrase** and by the final deliverables on shaping and discovery techniques/tools and on refactoring tools. In a sense, this concludes the design & tuning phase of parallel design patterns within **RePhrase**. The whole WP activities will be concentrated on the implementation final design and tuning from now on.

The main contributions to this deliverable come from UNIPI and UNITO (RePhrase patterns vs. general patterns, pattern implementation, RPL design), UC3M (pattern interface design), SCCH (advanced high level pattern design) and USTAN (pattern refactoring).

The following figures represent the relationships of this deliverables with the other WP deliverables and the other WPs (mainly) impacted by this deliverable.





Contents

Executive Summary 2
1.1 The pattern concept 4
1.1.1 Algorithmic skeletons 5
1.1.2 **RePhrase** patterns 6
1.2 Use of the Patterns in the Project. 7
1.2.1 RPL usage 8
1.2.2 Refactoring tool usage 8
1.2.3 “Manual” pattern usage 9
2.1 Initial Patterns 11
2.2 Advanced Patterns 14
2.3 Patterns Implementation 16
3.1 Programmability 19
3.2 Formal properties 21
3.3 Performance 22
3.3.1 The PARSEC suite 22
3.3.2 Pattern-based PARSEC implementation 23
3.3.3 Performance and programmability analysis 23

1. Introduction

This deliverable summarizes the experience within **RePhrase** concerning parallel patterns. In particular, it relates and frames the **RePhrase** achievements on the subject to the most general parallel design patterns research activities and concepts from the worldwide research community working in this area.

In this first part, we introduce the parallel design pattern concept, we describe the usage made of the patterns in the project and eventually we discuss **RePhrase** pattern exploitation with respect to what currently happens in other research and production contexts.

1.1 The pattern concept

The parallel design pattern concept dates back to the beginning of the century. The design pattern concept was already a well established and agreed concept at that time: a design pattern being intended as a common programming pattern solving a class of problems, using particular techniques, and eventually representing a kind of "recipe" a programmer may follow to solve his/her own instance of a problem belonging to the same class of problems solved by the pattern. The famous "gang of four book" [10] has been and it is one the worldwide texts used to teach software engineering students as well as programmers in general the philosophy of design patterns and the most common and useful object oriented design patterns in particular.

The parallel design pattern concept builds on top of the (non-parallel) design pattern concept, adding the "parallelism" dimension to patterns. The parallel design pattern concept has been strongly proposed and developed within the US software engineering community [15, 16].

A parallel design pattern may be intended as a pattern:

- to be used to solve a class of *parallel* problems
- to describe *state-of-the-art* general solutions and implementation techniques that can be used to solve the problem
- to outline the *forces* driving the pattern design process

- possibly, to provide examples of real code excerpts demonstrating the techniques to be used to solve the problem at hand.

The book [15] actually describes an additional step: parallel design patterns are organized into independent *design spaces*, each space hosting patterns solving problems at a different level of abstraction:

- the *finding concurrency* design space hosts patterns modelling the *kind* of parallelism exploited

- the *algorithm design* space hosts patterns modelling different typical parallel algorithms

- the *implementation structure* design space hosts patterns modelling different typical implementation designs, suitable to support different algorithm space patterns, and eventually

- the *implementation mechanisms* design space only hosts patterns modelling different mechanisms useful to implement the structures of the former design space.

1.1.1 Algorithmic skeletons

Dual to the pattern concept is the *algorithmic skeleton* concept, that actually dates back at least ten years before the software engineering design pattern concept. The algorithmic skeleton concept originated from the high performance computing community. An algorithmic skeleton represents a parametric, reusable and efficient implementation of a common parallel computation schema (we should say "pattern", actually) provided to the application programmer as an abstraction of the sequential programming language he/she is used to work with. Algorithmic skeletons may be in general freely nested to obtain more and more complex parallel computation schemes, or some nesting rules are provided stating that we can arbitrarily nest data parallel (stream parallel) skeletons within other data parallel (stream parallel) skeletons, but stream parallel skeletons should not be nested within data parallel skeletons (two-tier model) [14].

The algorithmic skeleton concept is dual to the parallel design pattern one in that it captures basically the same concept, but provides to the parallel application programmers ready-to-use programming abstractions (library calls in imperative programming, objects classes in OOP, higher order functions in functional programming) while parallel design patterns only provide "recipes" or–at best–sample parallel code to be adapted in programmer actual applications. We refer to *structured parallel programming frameworks* to both frameworks exploiting the parallel design pattern and algorithmic skeleton concept.

1.1.2 RePhrase patterns

In this project, following a trend recognizing the similarities in between the parallel design pattern and the algorithmic skeleton research activities [5], we adopted the approach of providing parallel patterns to the application programmers as programming abstractions that may be directly instantiated and used within application programmer code. In particular, since C++ is the "host" language adopted in **RePhrase** activities, patterns are provided in **RePhrase** as any other C++ "extension that is by actually providing patterns through libraries (see also Sec. 1.2).

This approach presents a number of advantages, with respect to classical, *non patterned* parallel programming frameworks, that we shortly will shortly discuss here below[1].

Programmability Parallel design patterns provided as sequential language primitives/abstractions greatly enhance *programmability*:

- the level of abstraction presented to the application programmer is raised, thus lowering the knowledge needed to write a parallel application,
- pattern implementation embodies most of the error prone details usually exposed to the application programmers, enabling the throughout implementation of well know optimizations directly within pattern implementation,
- the amount of code needed to implement complex parallel computations is greatly reduced,
- last but not least, application programmers may ignore a number of details that they have to consider when using non structured parallel programming frameworks provided (necessary *and sufficient* condition) they capture the informal parallel semantics associated to the parallel pattern.

Efficiency Application related and implementation related concerns are in the responsibilities of the parallel application programmer (providing the actual parallel application code), and the system programmer (providing the implementation of the patterns used by the application programmer). The application programmer, usually expert in the specific application domain, is only required to figure out which (combination of) parallel pattern(s) may be conveniently used to model his/her application. The system programmer, usually expert in parallel programming on the target parallel architecture, is only required to implement the parametric patterns using *state-of-the-art* solutions. This greatly improves efficiency:

- the application programmer may seamlessly experiment different alternative solutions that he/she only may figure out out of the application/problem to be solved at hand

[1] from now on, we will use the term *(parallel) patterns* to refer to patterns provided to the programmer as normal programming abstractions, that is as *algorithmic skeletons*

- the system programmer may seamlessly implement very complex implementation strategies ensuring compile and run time efficiency.

Code reuse Parallel design patterns are usually *parametric* with respect to the business logic needed to model the computation at hand. The business logic code has to be provided using some kind of "wrapper", in general, or using certain types already provided by the "host" programming language. In both cases, most of the code of an existing application may be re-used to build a parallel version of the same application, provided the original application has been properly designed and coded.

Maintainability Code maintainability is enforced by the modularity required by patterned code development. A parallel application is usually built out of the composition of several patterns, each with its own business logic code parameters. Both parallel patterns and business logic code units constitute units that represent the "object" of the maintainance process. The confinement of the strategies and interactions within the units ensure a better maintainability of the whole application code.

Portability Last but not least, portability is ensured via portability of the pattern compiler/run time support rather than by application code portability. The efficient implementation of the provided parallel patterns when targeting different architectures should guarantee that the application runs with comparable efficiency levels on the new architecture or, at least, that the application runs at the best efficiency supported by the used patterns on that kind of target architecture.

1.2 Use of the Patterns in the Project.

Within **RePhrase** we can identify three different usages of patterns as described in Sec. 1.1.2:

1. patterns may be used to design parallel applications *from scratch*, exploiting the RPL shell described in [6, 13, 18]

2. patterns may be introduced into existing sequential code using the refactoring tools developed within WP3 (see also [18, 20]

3. patterns may be used primitively in the use cases code to explore alternative solutions and to assess pattern implementation efficiency and proper refactoring schemes suitable to introduce the patterns into the sequential code right before these techniques get engineered in the refactoring tools of WP3.

In the following we detail the typical workflows relative to these three different usages.

1.2.1 RPL usage

The RPL shell is a prototype tool written entirely in C++ that supports a parallel application programmer in the *design* of a patterned parallel application.

The user may exploit the DSL provided by RPL shell[2] to seamlessly and automatically:

- define abstract pattern expressions and associate values to pattern non-functional properties (e.g. latency of business logic code)
- compose pattern expressions via patterns
- evaluate (via abstract models) non functional properties of pattern expressions (latency, service time, computing resources needed, etc.)
- apply known refactoring and optimization rules
- generate FASTFLOW code

The typical workflow is shown in Fig. 1.1 a).

Despite the fact the evaluators of non functional properties are based on abstract models, the "good" pattern composition among the set of equivalent (via refactoring) expressions may be easily identified by the programmer *before* moving to coding stage. This *de facto* shortens the time to production of the application. If we also consider the possibility to import existing business logic code into the shell and to generate automatically the parallel code (in FASTFLOW), the time to production of the application becomes order of magnitudes smaller than usual.

1.2.2 Refactoring tool usage

The Eclipse based refactoring tools developed within WP2 may be used to *parallelize an existing application* through the semi-automatic introduction of patterns. These tools support fundamentally refactoring actions built out of two distinct steps:

1. programmer selection of a portion of code to be "patterned", and
2. automatic refactoring of the selected code with the selected pattern.

These tools offer the possibility to generate Intel TBB, OpenMP, FastFlow or GrPPI patterned code.

The typical workflow relative to the usage of these refactoring tools is shown in Fig. 1.1 b). As for RPL, the **RePhrase** Eclipse based tools greatly reduce the development cost of applications parallelized through the introduction of patterns. The main effort required to the programmer/user consists in identifying the portions of

[2] RPL is a *textual*, *command line* shell

the code to be patterned. Then the whole code refactoring is managed automatically within the tool and the result is code that may be directly compiled with and linked to the target parallel programming framework. Within WP3 specific tools have been developed that identify in existing C++ sequential code different opportunities to introduce parallel patterns. These tools eventually support and simplify the unique step formally required to the programmer in this workflow. As a consequence, the time to production is reduced to values similar to the ones achieved when using the RPL shell.

1.2.3 "Manual" pattern usage

When developing the use cases, **RePhrase** beneficiaries have started experimenting using different patterns far before the refactoring tools were available. This means they used the target pattern libraries available "by hand", that is taking care manually of all the details needed to patternize an application (portion):

- identify the part of the application that has to be patterned
- properly wrap the relevant business logic code such that it may be reused in pattern parameters
- introduce the pattern call/instances in the original code, taking care of properly instantiating all the relevant parameters from
  - wrapped business logic code, or
  - non functional aspects of the pattern.

The typical workflow relative to manual patternization of existing applications is shown in Fig. 1.1 c).

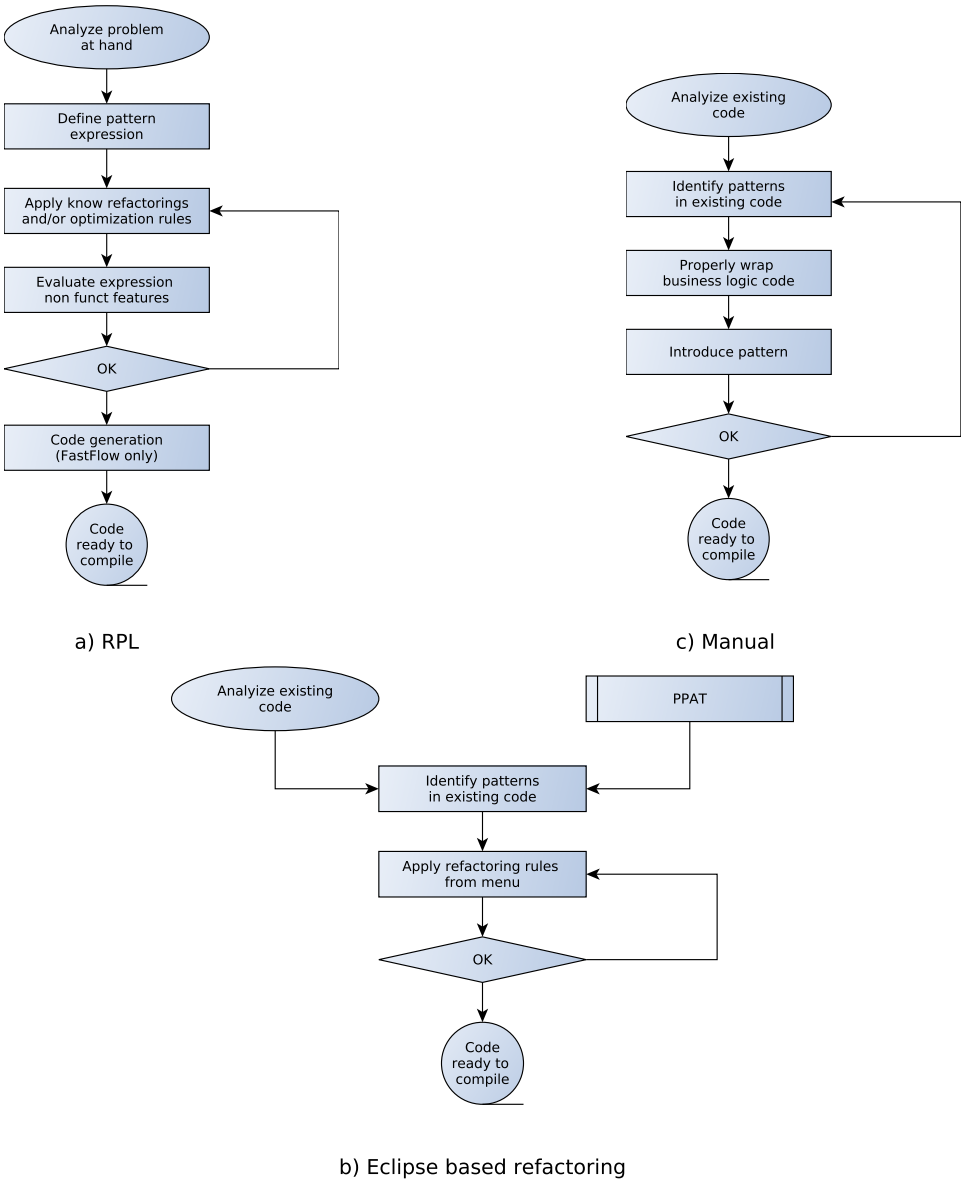


Figure 1.1: Typical workflows using patterns in **RePhrase**

2. The RePhrase Patterns Evolution

In this chapter we describe in a very general extent the evolution of the patterns definition during the project, starting from the initial patterns, which represent *standard* solutions available in most of the existing pattern-based parallel frameworks, up to the introduction of advanced patterns, which consist of an easy-to-use set of patterns oriented towards specific data-intensive kernels derived both from the use-case analysis and from the literature analysis. In addition, such patterns are designed in order to take into account the recent trends in Big Data and data-intensive applications.

2.1 Initial Patterns

The fundamental patterns are aimed at supporting the most common and practical data intensive parallel computations. They are named as *initial* because they represent the starting point for the design of a parallel application, which the developers use to sketch their initial parallel implementations. Also, they are used in the *core* software engineering related activities of the project, in order to focus on software engineering techniques targeting structured parallel software rather than generic parallel software. For their generality, they are not related to any of the project use cases, although realistically most of the advanced patterns rely on them as extensions, specializations or compositions of initial patterns.

According to the general parallel design pattern methodology [17], we identified a large set of initial patterns that belong to different broad categories:

- *sequential patterns*: they represent basic constructs to encapsulate sequential code in order to use the wrapped code as building blocks in the patterns recursive definition;

- *stream parallel patterns*: streams represent unbounded flows of data items, both in the case of primitive streams like data flows received from the networks, sensors, I/O devices, as well as in the case of fictitious streams generated by reading the elements of a large in-memory data structures. The aim

of this class of patterns is to increase throughput and consequently to reduce the completion time of the whole stream program;

- *data parallel patterns*: they model data-intensive computations applied on in-memory large data structures as a whole, i.e. processed by partitioning the data structures and by replicating processing functions.

These categories embrace all the consolidated trends in data-intensive computing, both batch-processing like in Apache Spark and MapReduce, as well as data streaming applications like in Apache Storm and IBM InfoSphere Streams.

The sequential patterns are a *wrapper* that wraps a sequential code portion such that it may be used to compute the application business logic in all those places where parallel patterns require a parameter expressing a computation. Two wrappers instances can be composed (*composition*) using the same resources, sequentially.

Tab. 2.1 lists the set of stream-parallel patterns with a short description for each. For each pattern the problem solved is also briefly summarized.

Name	Description
Farm	The 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.
Pipeline	The 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.
Filter	The 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) $\mathcal{P} : \alpha \rightarrow \{true, false\}$. The results of the filtering function of any stream data item must be independent from the other stream elements, i.e. it must be a pure function.
Accumulator	The 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 $\oplus : \alpha \times \alpha \rightarrow \alpha$, although commutative and associative functions will provide much better and more scalable implementation.
Stream Iterator	The pattern implements a function α stream $\rightarrow \alpha$ stream by iterating the computation of another pattern α stream $\rightarrow \alpha$ stream over one or more items appearing onto the input stream, and delivers results on the output stream.

Table 2.1: List of initial stream-parallel patterns developed in the project.

The project's patterns also include data-parallel patterns in the initial pattern set, because they are the *de facto* standard in most of the frameworks for *data*

intensive parallel computing (e.g., MapReduce and Apache Spark to cite some examples). Tab. 2.2 shows the patterns with a brief description for each.

Name	Description
Map	The 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.
Stencil	The 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.
Reduce	The pattern computes the result of the application of a binary (usually associative and commutative) function $\oplus : \alpha \times \alpha \rightarrow \alpha$ to all the data items of a collection of items of type α.
Map-Reduce	The 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 "reduction" of the values output for the same key in the first "map" phase.
Divide-and-Conquer	The 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 the solutions of the sub-problems.

Table 2.2: List of initial data-parallel patterns developed in the project.

In addition to such patterns, the initial pattern sets included also a set of patterns that do not have a real parallel semantics, but are rather useful to enable the possibility to compose stream parallel patterns and data parallel patterns within the same application, thus enabling a better degree of integration between the two pattern sets. In this way they are not two separated groups to be used disjointly. Tab. 2.3 lists the patterns that allow the generation of a stream (enabling stream parallel patterns to be used) or the generation of large data structures out of elementary elements (enabling data parallel patterns to be used).

Tab. 2.4 summarizes the patterns used for manipulating large collections of data items. They are designed to enable advanced strategies for data partitioning and collection, to be eventually executed in parallel.

Name	Description
Stream Generator	The pattern is in charge of producing a stream of data items all having the same type. Such sequence, potentially unbounded, can be obtained by reading the "external world", e.g., by receiving data items from a network connection, or it can be produced by unpacking a large in-memory data structure. The semantics of this patterns is to enable the possibility to instantiate stream parallel patterns from applications where a stream apparently does not exist.
Stream Collapser	The pattern receives data items from a stream and produces a single (collection) data structure out of them. The semantics allows the programmer or the project's refactory and pattern discovery tools to introduce data-parallel patterns in application scenarios where large data structures apparently do not exist.

Table 2.3: List of initial stream modifier patterns developed in the project.

Name	Description
Splitter	Split a collection into a set of possibly overlapping sub collections for further data parallel processing.
Merger	Merge a set of sub collections of data in a single collection after parallel processing.

Table 2.4: List of collection modifier patterns developed in the project.

2.2 Advanced Patterns

Advanced patterns are clearly characterizd in a specific usage context and are targeted to the parallelization of sequential (legacy) code. Examples are exploitation of loop parallelism, stream parallelism, data-parallel algorithms, execution of general workflows of tasks, etc. They are typically equipped with self-optimization capabilities (e.g. load-balancing, grain auto-tuning, parallelism-degree auto-tuning). They are considered advanced because of their fundamental impact both on the project's use cases and on the data-intensive application domains that are emerging as important research trends in the literature.

The first set of advanced patterns found during the corresponding project tasks are designed to model use cases of interest for the project's partners. They are:

- **Pool** pattern: the pattern models the evolution of a population of individuals. Iteratively, selected individuals are subject to evolution steps. The resulting new individuals are inserted in the population or discarded according to their "fitness" score. The process is iterated up to a given number of iterations (or up to a given computation time) or up to the point an individual with a given fitness is inserted in the population. Low fitness individuals may be removed from the population to keep the population size constant at each iteration;

- **Image convolution** pattern: the pattern computes the image convolution ac-

cording to some input "kernel" parameter. The image convolution is obtained from the source image processing each pixel at position i, j by taking a set of surrounding elements whose values are input of a kernel function that updates the value of the target pixel. Image convolution may be used to obtain different effects with different kernels, ranging from image blurring to image enhance, emboss, sharpen and so forth;

- **Stream Iterator with multiple outputs** pattern: it is a variant of the initial iterator pattern where we assume that at each iteration a value may be output onto the output stream–depending on the value of a "output guard" function–such that the cardinality of the stream may vary.

Furthermore, in order to accomodate the partners' requests, the pool pattern is designed according to two working modes: *synchronous* mode where iterations behave as barriers, that is all the evolution plus fitness computations are completed before considering the next iteration on the new population; *asynchronous* where individuals result of the evolution process are immediately evaluated and, in case, inserted in the population, asynchronously with respect to the processing of other individuals/evolutions.

Owing to the large diffusion of data stream processing (DaSP) frameworks [2] as effective programming solutions for data-intensive applications, a set of so-called DaSP patterns have been included in the project. Most of them use the so-called *windowing* abstraction widely diffused in stream processing, where a computation is applied every time a group of input data items is available and completely received. The semantics to define such groups can be extremely varying, with simple cases where the group size is fixed in cardinality (count-based windows) to cases where the items received in fixed temporal intervals (time-based windows) have to be processed as a whole. Essentially two patterns have been indentified in this set:

- **Windowed Farm**: the pattern is a specialization of the standard farm pattern from the initial set, where the workers are in charge of computing in parallel the user-defined function over different windows that are ready to be processed, while the emitter is in charge of distributing data items on-the-fly, eventually by transmitting a copy (reference) to the same item to multiple destinations. Finally, the collector is responsible to collect the results by eventually storing them back in the right order;

- **Keyed Farm**: also this pattern specializes the standard farm. Differently from the previous patterns, the assumption is that every input item can be deterministically assigned to a key identifier such that the physical input sequence is logically composed of multiple logical streams multiplexed together. The pattern exploits parallelism among items belonging to different keys, while data items of the same key must be executed by the same worker serially.

Finally, following the same rationale used for defining the initial patterns, the advanced set has beeen extended with advanced pattern modifiers in charge of allowing further possible compositions among advanced and initial pattern instances. Such patterns represent specializations of the patterns in Tab. 2.4 and 2.3. Tab. 2.5 summarizes such patterns called *basic building blocks* in the project terminology.

Name	Description
Inline stream generation pattern	This is regular stream generator pattern, where the optional input from the input stream is actually mandatory. Each of the original stream items may be used to generate a number of new items that are output onto the output stream.
Stream merger	The pattern accepts a number of input streams and produces an output stream hosting all the items appearing on the input streams, ordered according to a parameter policy.
Stream tupler	The pattern accepts a number of input streams and produces an output stream with tuples such that each tuple hosts items from the different input streams, built according to a parameter policy.
Stream splitter	The pattern generates a number of output streams out of a single input stream, according to a parameter policy.
Stream de-tupler	The pattern processes an input stream of tuples. Each tuple is used to generate items on different output streams according to a parameter policy.

Table 2.5: List of advanced stream modifier patterns developed in the project.

Fig. 2.1 outlines the different kind of patterns implemented within **RePhrase**.

2.3 Patterns Implementation

One of the main contributions of the project is the definition of a generic parallel pattern interface (GrPPI) fully compliant with the last C++17 standard. The interface, based on C++ template meta-programming techniques, allows the programmer to design pattern-based applications independently of the underlying backend and of the underlying computing platform, so by using and composing the identified parallel patterns by working at a sufficiently high level which is a pre-condition to reduce the time-to-development of parallel sotfware. Different backends have been considered in the projects: Intel TBB, OpenMP, FastFlow and a native implementation based on C++ threads. Furthermore, the interface is also supported for some of the patterns by a CUDA Thust backend, basically to support the execution of data-parallel patterns on heterogeneous systems.

While the implementation of the final set of advanced patterns with the GrPPI interface is under-development (it is the goal of deliverable D2.8), at the present time the project toolchain is equipped with a stable implementation for all the initial

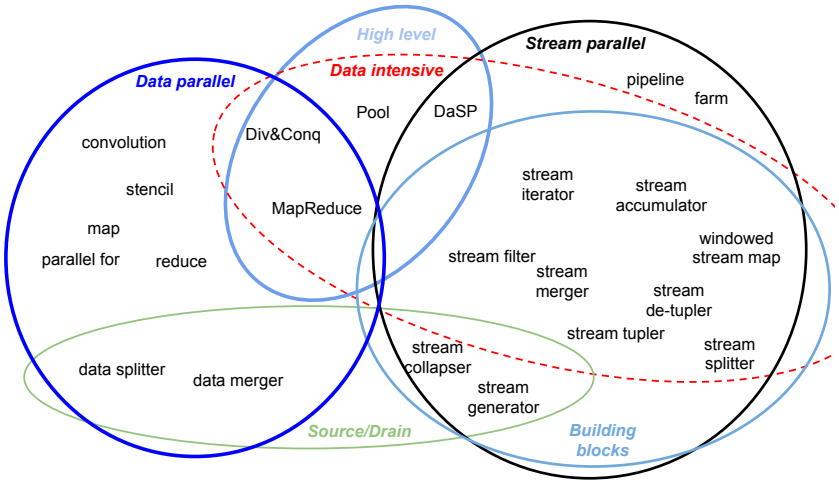


Figure 2.1: **RePhrase** patterns

patterns. Each initial pattern has been implemented using different backends as summarized in Tab. 2.6.

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

Table 2.6: Initial pattern set supported through **GRPPI** in different backends.

The table reports also in the last column the patterns directly provided using the FastFlow native interface. As we can observe, FastFlow is the only backend framework supporting all the initial patterns developed in the project while other backends (e.g., Intel TBB, OpenMP) support only a subset of them. This derives from the generality of the FastFlow framework, which represents the initial incubator of all the patterns developed in the project.

The initial performance analysis of the provided pattern implementation demonstrated that the performance of the patterns implemented directly with the backends compared with the one of the GrPPI implementation with the same backend is almost the same. As described in more detail in D2.4, the overhead introduced by

the abstraction layer adopted in the project is negligible while still allowing a easier parallelization process by the programmer by the automatic tools developed during the project. This demonstrated that the GrPPI interface represents a valuable tools and unified layer for the project's toolchain which allows efficient pattern instantiation on different architectures which makes very easy to change the underlying backends according to the application requirements.

Finally, the use of the pattern-based approach developed in the project has the merit to improve the so-called programmability, i.e. the programming effort in introducing parallel behaviors starting from a legacy sequential code. As summarized in D2.4, an analysis developed in a stream-oriented video streaming application showed that the number of lines added to the legacy code to introduce parallelism is very low (about 4.4%) while the attained performance is absolutely valuable and comparable with the one of using manually-written parallel software. To further confirm this trend in using the pattern-based methodology, which is the pillar of the Rephrase's approach, we have recently developed a more accurate analysis which will be summarized in this deliverable (see Sect. 3.3).

3. Parallel Patterns Payoff

This part of the deliverable aims at discussing the advantages deriving from the usage of patterns as observed in the **RePhrase** actitivites and from different perspectives: i) from the viewpoint of the programmability (level of abstraction exposed to the programmer(s) and time-to-deploy of patterned parallel applications) (see Sec. 3.1), ii) from the viewpoint of the code formal properties (e.g. correctness) (see Sec. 3.2 and iii) from the viewpoint of the performances achieved (see Sec. 3.3). Part of the results refer to activities related to **RePhrase** use case implementations and part refer to bechmarks, as the implementation of the project use cases is still ongoing at the moment being.

3.1 Programmability

In a sentence, the level of abstraction presented to the application programmer raises from the level of mechanisms to the level of policies. More classic, unstructured parallel programming frameworks basically provide the parallel application programmer with a set of mechanisms suitable to implement:

- concurrent/parallel/distributed activities setup,
- coordination among activities,
- communication among activities.

It's up to the application programmer to figure out how the single mechanisms can be used (adapted) to implement the parallel application they have in mind as well as the full setup/coordination/termination cycle of the needed concurrent/parallel/distributed activities. In some cases (see also Sec. sec:gli:altri) classic parallel programming frameworks provide components at a higher level of abstraction w.r.t. mechanisms. A notable example is the *parallel for* provided by OpenMP, IntelTBB, Cilk. Indeed, the parallel for may be intended as a parallel pattern but it lacks some of the peculiar features of patterns–such as compositionality–to be actually classified as a first class citizen of the parallel pattern set.

Structured parallel programming environments provide abstractions that can be understood as policies (strategies to orchestrate parallel mechanisms in a way they altogether implement a parallel pattern).

	Programming framework	
	Unstructured	**Structured (patterned)**
Components available	Activity setup, coordination and communication	Pattern instantiation, pattern composition
Coordination & communication	In charge to the application programmer	Implicit in pattern (nesting), in charge to the pattern programmer(s)
Design	Find concurrent activities, implement all of them out of the available mechanisms	Find concurrent activities, figure out which patterns may be used, instantiate patterns
Application debugging	All code interested: business logic code (functional debugging) and framework mechanisms code	Only business logic code interested
Performance tuning	May require changes in orchestration code (concurrent activity setup, coordination and communication)	Pattern parameter tuning (e.g. parallelism degree) or pattern expression tuning (alternative pattern composition, possibly provided automatically)
Target architecture knowledge	Required to application programmer)	Not required: hw targeting performed at the pattern implementation level
Parallel pattern implementation knowledge	Required to application programmer	Not required: inherited from system programmer's (the ones implementing patterns) work

Table 3.7: Different expertise and efforts required to program a parallel application (structured, i.e. patterned, vs. classic programming environments)

The change in the level of abstraction provided to the application programmer induces a change also in the expertise needed by the application programmers and in the amount of effort required to program a parallel application. These changes are qualitatively outlined in Tab. 3.7.

As a consequence, the time required to develop a parallel application using a structured (patterned) parallel programming environment may drop down by at least an order of magnitude, as already proven in other projects and research groups [1]. It is worthwhile pointing out once more that this improvement is a direct consequence of **both**:

1. the raise of abstraction level in the programming abstractions provided to the programmers; **and**

2. the availability of refactoring tools supporting patternization of code both existing and developed from scratch.

Patterns and Use Cases

Within **RePhrase**, different use cases and applications have been identified to assess the project results [19] including applications for:

Use case	Pattern(s) opportunities
Railway diagnostic systems	MapReduce, Stencil, (Pipe & Farm)
Medical image processing	Map (or Parallel For), Stencil, Pipe & Farm
Stochastic local search	Map, Pool Evolution (synch & asynch versions)
Weather forecast	Map, Reduce, Stencil

Table 3.8: Patterns in use cases

- stochastic local search
- weather forecast
- railway diagnostic system
- medical image processing

Within [22] the opportunities for parallelism exploitation have been investigated relatively to all these applications and considering the range of parallel patterns provided by WP2 within **RePhrase**. The relative findings are outlined in Tab. 3.8.

3.2 Formal properties

Pattern implementation has been subject to the application of the **RePhrase** software engineering methodology to individuate possible errors and mistakes in the implementation of the patterns. This process (documented since [21]) enables the verification of formal properties of the parallel code at the level of the implementation of the patterns. In case the application programmer only used patterns or pattern nestings to model and implement the whole parallel behaviour of his/her application, the verification of the formal properties in the pattern implementation *de facto* ensures that the parallel application will run correctly as far as parallelism is concerned.

The usage of the check tools on FASTFLOW and Intel TBB libraries, two of the back-ends used within **RePhrase** to provide and support patterns, spotted a few problems that will be eventually fixed before the end of the project. It is worth pointing out that the fix of these problems

- will be a completely independent activity w.r.t. to programming effort actually related to the implementation of the use cases,
- will be conducted by independent programmers (the pattern run time support developers) w.r.t. use case application programmers, and
- will be independent of the programming efforts spent within the project to provide the **RePhrase** refactoring and pattern discovery tools.

3.3 Performance

In this section we describe a first performance assessment for the pattern-based parallel methodology developed in the project. Our work consists in a complete porting of all the parallel applications of the PARSEC benchmark suite (the Princeton Application Repository for Shared-Memory Computers) [3] using parallel patterns. The goal of this assessment is to show that our methodology, based on reusable and composable parallel patterns, is expressive enough to model real-world complex applications, and that the programmability effort can be greatly reduced compared with using standard parallel programming frameworks. Since the applications of the suite are absolutely realistic, and widely studied and applied as benchmarks in past research work, this represents a valuable assessment of the methodology underneath the project toolchain.

3.3.1 The PARSEC suite

PARSEC consists of 13 programs from different areas of computing. Each application is provided with several input sets for each benchmark. Three datasets, with different sizes, target the execution on simulators (i.e. *sim-small*, *sim-medium*, *sim-large*), while the *native* dataset is representative of a realistic execution scenario of the application.

From the parallel programming perspective, PARSEC applications are of great interest for testing frameworks because they have different memory access behaviors, data sharing patterns, amount of parallelism, computational granularity, and synchronization frequency. Table 3.9 reports the official name of the benchmarks and their parallelism model. Furthermore, we show the official parallel versions released within the PARSEC suite.

Benchmark	**Domain**	**Parallelism**	**Parallel Versions**		
			Pthreads	*OpenMP*	*Intel TBB*
`blackscholes`	Financial Analysis	data par.	✓	✓	✓
`bodytrack`	Computer Vision	data par.	✓	✓	✓
`canneal`	Engineering	unstructured	✓	✗	✗
`dedup`	Enterprise Storage	stream	✓	✗	✗
`facesim`	Animation	data par.	✓	✗	✗
`ferret`	Similarity Search	stream	✓	✗	✗
`fluidanimate`	Animation	data par.	✓	✗	✓
`freqmine`	Data Mining	data par.	✗	✓	✗
`raytrace`	Computer Vision	data par.	✓	✗	✗
`streamcluster`	Data Mining	data par.	✓	✗	✓
`swaptions`	Financial	data par.	✓	✗	✓
`vips`	Media Processing	data par.	✓	✗	✗
`x264`	Media Processing	stream	✓	✗	✗

Table 3.9: Classification and characteristics of the PARSEC v3.0 applications.

Most of the applications belong to the data parallelism model, where the computation is performed on large data structures logically partitioned among multiple threads. Stream parallelism characterizes applications where a large sequence of data items are processed by a chain of threads that execute distinct computation phases on different items in parallel and in a pipeline fashion. The case of `canneal` is an example of applications that do not straightforwardly follow any common parallelism paradigm (in the table it is referred to as *unstructured*).

3.3.2 Pattern-based PARSEC implementation

Starting from the Pthreads implementations available in the PARSEC suite, we designed and implemented a parallel version of each application by composing and nesting the patterns that originally belong to the initial patterns set developed in the project. All the PARSEC applications, except x264, have been parallelized by composing the patterns of the project implemented in FastFlow. For some of the data-parallel patterns, we also implemented a version using SkePU [9], a pattern-based framework for data-parallel applications developed outside the project. The choice to adopt also an external framework is to assess the differences between the tools that we are developing in the project and other counterparts that exist in the scientific community with a well-assessed credibility although limited to a subset of the patterns available (e.g., only data-parallel ones).

Tab. 3.10 summarizes for each PARSEC application which patterns have been used and a brief description by words of how thay have been nested or composed. A more precise description of the pattern-based implementation is described in [7].

3.3.3 Performance and programmability analysis

In this part we summarize the analysis explained in detail in [7]. We first focus on the performance achieved by the pattern-based parallel implementation of the PARSEC applications compared with the implementations already supported by the suite, see Tab. 3.9. Furthermore, to provide a more complete analysis we also analyze the performance and programmability effort of using a task-based parallel programming model, which is a widely diffused appraoch to parallel programming. We use ompSs [4], a framework based on pragma annotations to express general graph dependencies between tasks that are executed by the underlying parallel data-flow execution model. We use the work published in [4] for the comparison, where almost all the PARSEC applications have been already implemented using ompSs.

In terms of programmability, we measured two metrics:

- **Lines of Code**: this metric is commonly used in software engineering to measure code and programming complexity. For each benchmark we considered only the source files required to implement the parallelization or modified during the parallelization (the other files are the same in all the

Benchmark	Patterns used	Description
`blackscholes`	iterator, map	This benchmark is an iterative data-parallel computation. We model it as an iterator pattern where the internal pattern is a map.
`bodytrack`	iterator, map	The benchmark is implemented as a outermost iterator pattern, where the internal pattern is the sequential composition of the inner iterator instances, each one applying a set of maps.
`canneal`	master-worker	The master-worker pattern is written in FastFlow as a variant of the farm pattern where the collector is eliminated and the emitter is in charge of distributing, also many times util a condition is verified, the same input task to a corrisponding worker according to a load balancing policy.
`dedup`	pipe, farm	The application is modeled as a pipelien of five stages, where the first and the last ones are sequential while the others are farm. This implementation is the closest to the original Pthreads version.
`facesim`	iterator, map	The application is modeled as a complex composition of iterators and maps nested together. Totally, we recognized 19 maps where some of them are executed iteratively until a global condition is met.
`ferret`	pipe, farm	Similar to dedup.
`fluidanimate`	iterator, map	The application is an iterator pattern where the inner pattern is a sequential composition of nine parallel maps.
`freqmine`	map, iterator	Composition of seven maps where the last is iterated many times.
`raytrace`	iterator, map	Simple iterator pattern with inside a single map.
`streamcluster`	iterator, map	As for facesim, the application is an iterator of several sequential maps and innermost iterators of maps.
`swaptions`	map	Single map.
`vips`	pipe, farm	The application is modeled as a pipeline of two stages, where the first one is a farm of a sequential pattern.

Table 3.10: Pattern-based implementations of the PARSEC applications.

versions). These files include the definition of data structures used for thread communications, synchronization mechanisms and the files containing calls to the different parallel programming frameworks. The measures have been normalized with respect to the Pthreads version (i.e. Pthreads is always 1, a value greater than 1 means more lines of code and lower than 1 means fewer lines of code);

- **Code Churn**: it is defined as the number of lines modified and added with

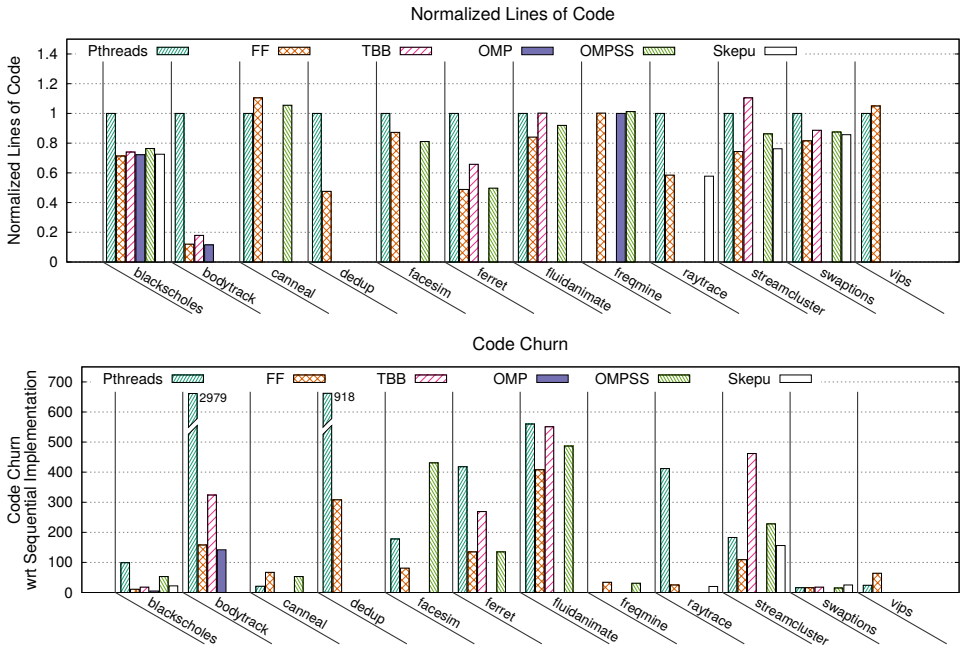


Figure 3.2: **(a)** Lines of Code (LOC) of the different parallel implementations, normalized between 0 and 1 (the lower the better). **(b)** Code Churn (i.e. number of modified and added lines of code) of the different parallel implementations with respect to the original sequential implementation (the lower the better).

respect to a previous version. In our case we consider the code churn of each parallel version with respect to the original sequential code. Starting from the sequential code, two different parallel implementations may have a similar number of code lines. However, if an implementation needs to modify and to introduce a higher number of lines, this means that the effort required is likely higher than the one needed to implement the other versions. This metric is computed on the files normalized with the same process described for Lines of Code.

We analyze the two metrics over all the benchmarks and over all the parallel versions. The results are shown in Fig. 3.2(a) and 3.2(b).

In terms of performance instead, the original work was developed on three different classes of multicore architectures. In this deliverable we report the results achieved on an Intel Xeon Server only, a dual-socket NUMA machine with two Intel Xeon E5-2695 Ivy Bridge CPUs running at 2.40GHz featuring 24 cores (12 per socket). Each hyper-threaded core has 32KB private L1, 256KB private L2 and 30MB of L3 shared with the cores on the same socket. The machine has 64GB of DDR3 RAM. Linux 3.14.49 x86_64 shipped with CentOS 7.1. The available compiler was `gcc` version `4.8.5`.

The performance results are shown in Fig. 3.3, where the best execution times

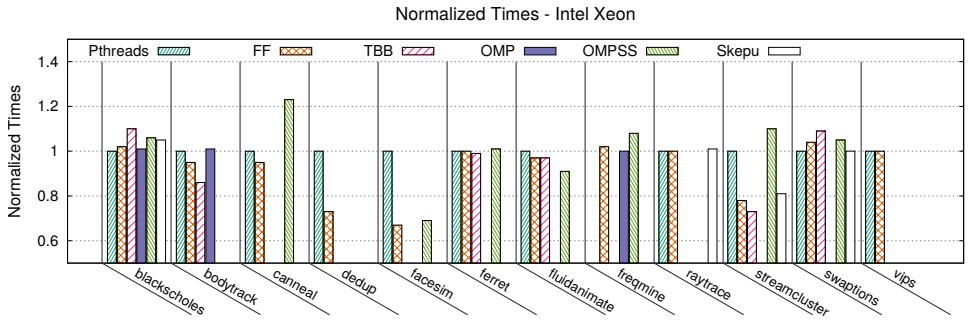


Figure 3.3: Best execution times normalized with respect to the PARSEC reference (i.e. OPENMP for `freqmine`, Pthreads for the remaining benchmarks). Intel Xeon server architecture.

of the benchmarks for each version, obtained by varying the number of threads, have been normalized to the PARSEC reference implementation (i.e. openMP for `freqmine`, Pthreads for all the others benchmarks). Accordingly, values lower than 1 represent cases with execution time lower than the one of the reference PARSEC implementation. Small differences and discrepancies in the results (between different versions of the same benchmark) are reasonably due to differences in the compiler and by the intrinsic differences and optimizations in the runtime of the frameworks used.

To summarize the results, we achieved an average reduction of 26% in the lines of code (in both FASTFLOW and SKEPU) compared with the original Pthreads implementation, and an average reduction of 3% with respect to the OMPSS implementations. In the best case, we reduced the lines of code up to 87% with respect to Pthreads, and 14% compared with the OMPSS versions. The code churn is in average 58% lower than Pthreads and 34% lower than OMPSS version. Concerning the performance, the FASTFLOW implementations obtained an average performance gain of 14%, with a maximum gain of 42% and a maximum loss of 9% with respect to the Pthreads one. Considering the benchmarks implemented with SKEPU, we obtained an average performance gain of 7% (maximum gain of 45%, maximum loss of 11%). Finally, OMPSS implementations obtained an average gain of 2% (maximum gain of 37%, maximum loss of 23%) with respect to the Pthreads implementation.

4. Positioning

The pattern methodology adopted and pushed forward by **RePhrase** actually relies and builds on results available from different research projects and groups. In particular, we inherited from several EU funded projects that were funded in the past years, including:

CoreGRID This Network of Excellence (2005–2008), funded in the FP6 program framework. Within the NoE, the WP3 "Programming model Institute" coordinated the activities that eventually led to the first idea of GMC, the Grid Component Model encompassing the idea of having pattern components to implement parallel/distributed computations targeting grids (http://coregrid.ercim.eu/mambo/).

GridCOMP The FP6 STREP project (2006–2009) was a kind of *spinoff* project of the above mentioned CoreGRID NoE aimed at providing a working and efficient implementation of GCM. The implementation was eventually provided on top of ProActive, the parallel/distributed framework supported at that time by INRIA Sophia Antipolis and by the Univ. of Nice (F) (http://gridcomp.ercim.eu/).

ParaPhrase The FP7 project (2013-2016) contributed to the development of the FASTFLOW pattern library targeting multi-core and GPU equipped hardware workstations (http://www.paraphrase-ict.eu).

REPARA The FP7 project (2014-2017) contributed to the development of autonomic pattern management techniques supporting the dynamic selection of different targets for different parts of the pattern expression implementations (http://repara-project.eu/).

Peppher The FP7 project "Performance Portability and Programmability for Heterogeneous Many-core Architectures" uses pattern to efficiently target heterogeneous architectures (http://www.peppher.eu/).

EXCESS The FP7 project "Execution Models For Energy-Efficient Computing Systems" uses patterns to achieve energy efficiency while targeting heterogeneous architectures (multi-core with accelerators, http://www.excess-project.eu/index.html).

Parallel patterns are not an exclusive of EU research projects, indeed. Several well know parallel programming frameworks start to include pattern concepts and assessments. As an example:

Intel TBB (https://www.threadingbuildingblocks.org/) provides parallel for (map), reduce and pipeline patterns, that can be nested to implement more complex parallel patterns. The "Threading Building Blocks Design Patterns" [12] document mentions how several other patterns may be implemented using primitive TBB components.

OpenMP (http://www.openmp.org/) provides since the very beginning the parallel for (map) pattern. More recent versions introduced the possibility to implement reduce pattern and to use the task concept to implement several other patterns, that are not provided natively, indeed.

Microsoft PPL (the Parallel Pattern Library, https://msdn.microsoft.com/en-us/library/dd492418.aspx) provide both stream and data parallel patterns in addition to several low level mechanisms that may be used to implement and integrate your own parallel patterns in case these are not available within the library.

Google MapReduce ([8]) provide very efficient implementation of mapreduce pattern targeting distributed clusters and processing massive amounts of data.

C++ STD (https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html) provides parallel implementation of several "algorithms" that may be actually considered parallel patterns, particularly within the C++17 standard.

In addition to these frameworks, we should mention a number of research frameworks that currently provide and maintain different pattern/skeleton based parallel programming frameworks, including:

FastFlow (http://calvados.di.unipi.it/) by Univ. of Pisa and Torino, providing nestable stream and data parallel patterns as C++ classes and mainly targeting shared memory multi-cores with accelerators

SKEPU (http://www.ida.liu.se/labs/pelab/skepu/) by Univ. of Linkoping, providing mainly data parallel patterns as C++11 classes and targeting (multiple) GPUs

SkeTo (http://sketo.ipl-lab.org/) by the Univ. of Tokio, providing nestable data parallel patterns as C++ classes

SkeCL ([23]) is an high level skeleton library targeting (multiple) GPU(s).

SPAR ([11]) uses annotations to provide high level stream parallel patterns in C++.

Overall, the effort made in **RePhrase** to adopt parallel patterns exploits the results from a number of these existing or past research activities but also adds new dimensions in the research area:

- the principal rôle of refactoring as the main way to introduce parallel patterns into code.
- the principal rôle of software engineering to prove parallel pattern implementation correctness far before the patterns are actually used *and* patterned application correctness.
- the introduction of specific parallel patterns (e.g. the asynchronous pool evolution pattern).
- the introduction of a general purpose parallel pattern interface, fully compliant with C++ standards *and* supporting multiple back-ends.
- the introduction of new and automatic techniques to identify portions of sequential applications that may be efficiently parallelized using parallel patterns.
- the impact on standardization bodies, in particular the C++ standard committee, aiming at introducing patterns into language standard libraries.

5. Conclusions

This deliverable provides a) the final **RePhrase** pattern set and b) an evaluation of the pattern related achievements in **RePhrase**.

We started re-calling the general concept of parallel design pattern and the **RePhrase** perspective adopted with respect to patterns and algorithmic skeletons.

Then we described the typical usage/exploitation made of the parallel design pattern(s) within **RePhrase** and we summarized the patterns included in the final **RePhrase** pattern set. The set is actually the very same already discussed in [20], as after that deliverable we did not come into the necessity of designing new patterns and including them in the project pattern set. The only minor change has been related to the asynchronous pool evolution pattern, required to optimize the implementation of one of the use cases (stochastic local search).

Eventually, we provided a quick overview of other research activities–all related to parallel design pattern research–run in the recent past or currently being run by different groups, and we outlined the main different perspectives introduced withing **RePhrase** that differentiate the project research from these related research activities.

Bibliography

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Carlo Montangero, and Laura Semini. Managing adaptivity in parallel systems. In *Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers*, pages 199–217, 2011.

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

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[4] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Trans. Archit. Code Optim.*, 12(4):41:1–41:22, December 2015.

[5] Marco Danelutto. On skeletons and design patterns. In G.R Joubert, A. Murli, F.J. Peters, and M Vanneschi, editors, *Parallel Computing: Advances and Current Issues (Proc. of Intl. ParCo 2001)*, pages 425âĂŞ–432. Imperial College Press, London, 2001.

[6] Marco Danelutto, Massimo Torquati, and Peter Kilpatrick. A DSL based toolchain for design space exploration in structured parallel programming. In *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, pages 1519–1530, 2016.

[7] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. Bringing parallel patterns out of the corner: the p^3arsec benchmark suite. *ACM Transactions on Architecture and Code Optimization*, 2017.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[9] 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.

[10] 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.

[11] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. Spar: A DSL for high-level and productive stream parallelism. *Parallel Processing Letters*, 27(1):1–20, 2017.

[12] Intel. Intel Threading Building Blocks Design Patterns, 2010. https://software.intel.com/sites/default/files/m/4/8/1/e/e/33963-Design_Patterns.pdf.

[13] Vladimir Janjic, Christopher Brown, K. Mackenzie, Kevin Hammond, Marco Danelutto, Marco Aldinucci, and José Daniel García. RPL: A domain-specific language for designing and implementing parallel C++ applications. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 288–295, 2016.

[14] Herbert Kuchen. A skeleton library. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 620–629, London, UK, UK, 2002. Springer-Verlag.

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

[16] 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.

[17] Jorge Luis Ortega-Arjona. *Patterns for Parallel Software Design*, volume 21. John Wiley & Sons, 2010.

[18] RePhrase. Report on the initial pattern set, 2015. Deliverable D2.1 of WP2, RePhrase project, https://www.rephrase-ict.eu.

[19] RePhrase. Requirements on use cases, methodology, and tools, 2015. Deliverable D6.1 of WP6, RePhrase project, https://www.rephrase-ict.eu.

[20] RePhrase. Advanced patterns, 2016. Deliverable D2.5 of WP2, RePhrase project, https://www.rephrase-ict.eu.

[21] RePhrase. Combined report describing testing, verification, catastrophic failures detection and properties violation detection for the initial set of patterns, 2016. Deliverable D3.2 of WP3, RePhrase project, https://www.rephrase-ict.eu.

[22] RePhrase. Report on defined Use Cases, 2016. Deliverable D6.3 of WP6, RePhrase project, https://www.rephrase-ict.eu.

[23] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl - a portable skeleton library for high-level gpu programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1176–1182, Washington, DC, USA, 2011. IEEE Computer Society.