Project no. 644235

REPHRASE

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

Report on the initial pattern set
D2.1

Due date of deliverable: Setp. 30, 2015 (M6)

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

Responsible institution: UNIPI
Editor and editor's address: M. Danelutto, 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)	

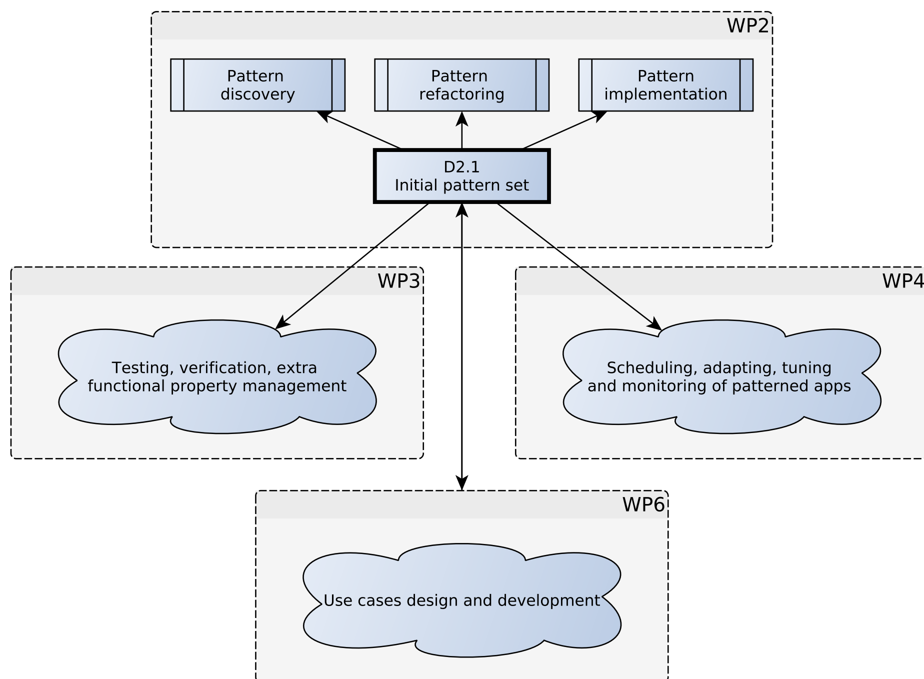
Change Log

Rev.	Date	Who	Site	What
1	30/09/15	Marco Danelutto	UNIPI	Original Version
2	26/09/16	Kevin Hammond and Vladimir Janjic	USTAN	Review (syntax fixes)
3	20/09/17	Marco Danelutto	UNIPI	Added changelog, reformatted placement picture in the executive summary

Executive Summary

The deliverable is the initial deliverable from WP2 “Pattern-Based Parallel Software Engineering”. It hosts i) the initial, high level design of the **RePhrase** pattern set and ii) a preliminary outline of the **RePhrase** parallel pattern DSL. The deliverable is the result of the first phase of WP2 (and Task 2.1, “Patterns for Data-intensive and Parallel Applications”) where, according to the DoW *we will identify and develop a fundamental set of commonly used patterns found in C++ applications (such as map-reduce, farm and pipeline), together with the part of the DSL that describes them*. As shown in the Figure, the deliverable contents impact on the rest of the activities of WP2, provide inputs to the activities in WP3 aimed at ensuring reliability, robustness and integrity of structured parallel software, and constitute a kind of base target framework for the adaptivity strategies and techniques developed in WP4.

The contributions to the deliverable can be summarized as follows: UNITO (pattern description and semantics), UNIPI (RPL), UC3M (pattern interface) and USTAN (pattern refactoring).



Contents

Executive Summary 2

Introduction **5**

2.1 **RePhrase** structured parallel programming methodology 7

RePhrase vision of the patterned application design process **8**

3.1 Sequential patterns 11

3.1.1 Sequential code wrapper 11

3.1.2 Sequential composition 12

3.2 Stream parallel patterns 13

3.2.1 Farm 13

3.2.2 Pipeline 14

3.2.3 Stream filter pattern 15

3.2.4 Stream accumulator pattern 16

3.2.5 Stream iteration pattern 17

3.3 Data parallel patterns 18

3.3.1 Map 18

3.3.2 Stencil 19

3.3.3 Reduce 20

3.3.4 MapReduce 21

3.3.5 Divide and conquer 22

Data collection management **23**

4.1 Streams 24

4.1.1 Stream generator pattern 24

4.1.2 Stream collapser pattern 25

4.2 Data collections 26

4.2.1 Splitter 26

4.2.2 Merger 26

Stateful patterns **27**

Pattern composition rules **29**

RePhrase pattern DSL **30**
7.1 High level pattern language 32
7.2 Annotations 32
7.3 Rewriting rules 33
7.4 (Non functional) property models 34
7.5 Optimizers 34
7.6 DSL implementation 35

Conclusion **35**

1. Introduction

One of the main aims of **RePhrase** WP2 is to provide the application programmers with a comprehensive set of parallel patterns that may be used to implement efficient parallel applications.

The parallel patterns will be particularly aimed to support *data intensive* applications, that is applications that, while non being necessarily characterized as “big data”, process non negligible amounts of data. These data will be either becoming available at different times or being available all-together at the beginning of application execution. In the former case, proper *stream parallel* patterns will be exploited to orchestrate and implement the application parallel behavior. In the latter, proper *data parallel* patterns will be used instead.

This document introduces the initial set of fundamental parallel patterns supporting the most common and practical data intensive parallel computations. This set represents the initial set of patterns, used by the use case developers in the project to sketch initial parallel implementation of their applications. Also, they will be used in the “core” software engineering related activities of the project to focus generic software engineering techniques on *structured* parallel software rather than on generic parallel software.

Later on in the project, this initial pattern set will be integrated with more domain and data intensive specific patterns suitable to better support the project use case development but general enough–in the full respect of the (parallel) design pattern concept–to support a large number of parallel applications.

Eventually, this deliverable will sketch the main features of the high level DSL we will use within **RePhrase** to enable and support application programmer in the high level parallel design of their applications. As stated in the **RePhrase** DOW

> The DSL will enable patterned applications to be designed without the application programmer having to deal with low-level implementation details. It will provide a number of different features, including a) the ability to efficiently and concisely represent all the patterns, b) the ability to represent all the pattern information needed for the efficient implementation that will be developed in T2.2, and c) the ability to support pattern rewriting techniques designed in T2.5, so providing application programmers with the possibility to experiment with different patterns/pattern compositions at design time.

Being this document an "early" **RePhrase** deliverable (due 6 months after the project start), the DSL is only described in terms of its main features. It will be fully designed and implemented later on in the project and described in forthcoming WP2 deliverables, in particular in D2.4 (Software for pattern implementations for the initial pattern set) and D2.8 (Software for pattern implementations for the advanced pattern se), related to the pattern framework implementation, and in D2.7 (Final report on patterns and relationship with general design patterns), related to the final pattern framework features.

The report contents are organized into three main parts:

- Chapter will shortly introduce the structured parallel programming design methodology enforced by the extensive usage of parallel design patterns to orchestrate the application parallel activities.
- Chapters 2.1 to 4.2.2 will introduce the different components of the initial pattern set
- Eventually, Chapter 4.2.2 will introduce the main features of the **RePhrase** parallel pattern DSL.

2. Patterned application design

In this Chapter, we briefly recall the structured parallel programming methodology adopted within **RePhrase**, with the aim to introduce the scenario where the patterns described in Chap. 2.1 and the DSL described in Chap. 4.2.2 are actually used.

2.1 RePhrase structured parallel programming methodology

Within **RePhrase** we assume to adopt a *structured* parallel programming methodology supporting parallelism exploitation via *parallel design patterns* and parallel design pattern *compositions*.

A parallel design pattern, as discussed in [9], *is simply a generalizable solution to a recurring problem that occurs within a well-defined context*. In a sense, a parallel design pattern can be understood as a "recipe" eventually supporting the parallel application programmer in the development of parallel applications where parallelism exploitation is achieved according to the schema modelled by the pattern. Parallel design patterns originated within the software engineering community [11–13] after the object oriented patterns as described in the "gang-of-four" book [7] and have been advocated to be a viable solution to overcome the "software gap" [2].

Recently, the idea of implementing parallel design patterns through algorithmic skeletons, in such a way design recipes may be turned into actual programming abstractions has been explored [4, 8]. An *algorithmic skeleton* is a parametric, customizable, efficient and reusable *implementations* of parallel patterns directly provided to the parallel application programmers according to the principle of "minimal disruption" stated by M. Cole [3].

RePhrase adopts the idea that parallel applications should be developed by properly designing its parallel structure according to parallel design patterns or pattern compositions and then implementing the pattern composition exploiting the available algorithmic skeletons. This both in case the parallel application is developed from scratch and in case it is designed as parallel refactoring of an existing sequential application.

In the two cases, the parallel structure design is assumed to take place using a proper parallel design pattern DSL supporting the user in the exploration of the

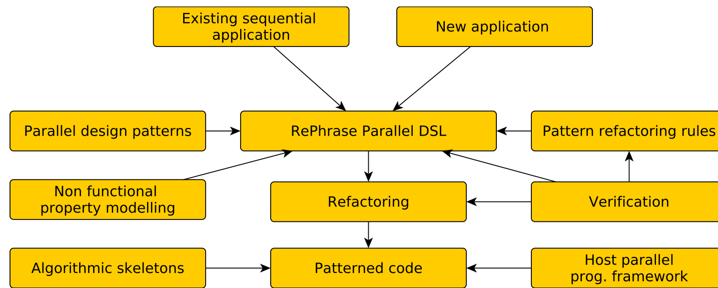


Figure 2.1: **RePhrase** vision of the patterned application design process

space of possible parallel implementations of the application at hand. The existence of a "space of possible parallel implementations" comes from the possibility to exploit different algorithms as well as different parallel implementation schemas.

The usage of an high level DSL suitable to represent different alternative implementations of the same application and to evaluate (suitable approximations of) their non functional features (asymptotic or ideal performance, energy consumption, efficiency, scalability, fault tolerance, security, etc.) greatly improves the possibilities given to the parallel application programmers to experiment and comparatively evaluate different parallel implementations for the application at hand.

Eventually, the DSL representation of the parallel structure of the application may be directly used to drive the process of actual code generation, possibly exploiting proper refactoring techniques. Figure 2.1 outlines the **RePhrase** vision of the design process of patterned parallel applications.

More in detail, let's shortly introduce the different parts of the "vision" picture.

Parallel design patterns host the "palette" of patterns available to the parallel application programmers to model the parallel behavior of their applications. The actual patterns included in the initial set, the one available at M6 in the project timeline to start programming activities is the main subject of this deliverable.

RePhrase parallel DSL is the language supporting the parallel application developed in the design of the parallel structure of their applications. DSL provides mechanisms to *represent* parallel patterns and parallel pattern compositions, to *evaluate* non functional properties of the pattern expressions according to suitable, high level non functional property models, and to *transform* pattern expressions into functionally (only)[1] equivalent pattern expressions.

Pattern refactoring rules are the well known or **RePhrase** specific rewriting rules that may be used to transform a pattern expression into a functionally equivalent, different pattern expression. Functional equivalence has to be intended as

[1]that is computing the same results, with different non functional features/properties

the property ensuring both expression eventually compute the same result, possibly with different non functional features (performance, energy consumption, efficiency, etc.).

Non functional property models provide approximate, high level evaluation of the non functional properties of DSL parallel pattern expressions. Analytical or soft computing models may provide the expected service or completion time of the pattern (composition), an approximation of the energy used to compute the result, the theoretical efficiency of the pattern, etc.

Algorithmic skeletons provide ready to use, efficient, parametric and customizable implementations of parallel design patterns.

Refactoring activities are used to actually produce the application code out of the DSL patterned specification of the application and of the algorithmic skeleton available.

Verification activities ensure appropriateness and correctness of the pattern rewriting rules as well as of the overall factoring process leading to the actual application code.

3. Fundamental patterns

This part of the document introduces the patterns we will consider in the first part of **RePhrase**. We call these parallel design patterns "fundamental" as we aim at including in the initial set the most common and throughly used known parallel design patterns.

The description provided of the patterns does not include an API, although for each one of the patterns the main functional and non functional "parameters" are outlined. The complete API for the patterns will be provided in D2.4, where the initial implementation of this pattern set will be provided on top of different parallel programming frameworks such as FastFlow, OpenMP or OpenCL.

Also, the pattern description will be given in a synthetic way, pointing to existing references for more comprehensive and detailed description of the patterns, as the main aim of this document is to list the different alternatives provided within the **RePhrase** initial pattern set, such that use case developers may start figuring out patterned parallelisation strategies of their applications.

The initial set of **RePhrase** patterns will be 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'll also list some "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

Section	**Description**
Also known as	introduces synonyms for the pattern name
Problem solved	describes the abstract problem (pattern) solved (implemented)
Functional parameters	the parameters characterizing **what** is computed by the pattern
Non functional parameters	the parameters characterizing **how** the functional results of the pattern are computed
Implementation	hosts some general view of possible implementations and of the related problems that a system programmer has to deal with in the pattern implementation
Functional semantics	describes which results is computed out of each initial data
Parallel semantics	describes what is actually computed in parallel within the skeleton
Component pre-conditions	describes which pre-conditions must be checked true on the pattern components to be able to use it as pattern

Table 3.1: Pattern description fields

functional parameter of other patterns (e.g. a stage in pipeline or a worker in a map pattern).

For each one of the patterns considered, we will provide a design pattern description with the fields described in Table 3.1.

3.1 Sequential patterns

3.1.1 Sequential code wrapper

Problem solved The pattern wraps a sequential code portion such that it may be used to compute the application "business logic" in all those places where parallel pattern require a parameter expressing a computation.

Functional parameters The code to be wrapped, along with its interface (the input and output parameters needed).

Non functional parameters Non functional parameters of interest in the sequential wrapper pattern may include libraries and source file dependencies as well as the tools (and tool parameters) needed to compile and link the file when non standard tools are actually necessary.

Implementation The implementation usually require to wrap the business logic code in a function declaration, accepting as parameter a data structure hosting the whole set of the input parameters and returning a data structure hosting the whole set of output parameters. Possibly, components of these two data structures (data types) may be pointers, if shared memory is supported on the target (hw/sw) architecture.

Functional semantics A sequential code wrapper pattern encapsulates a code ideally computing a function f out the α typed input parameters and providing as a results a β typed result. We will therefore assume the semantics of the wrapper may be defined by formally providing the function $f : \alpha \to \beta$. A sequential wrapper pattern acting as a parameter in a stream parallel pattern or composed to other stream parallel patterns will behave as a stream parallel patterns, that is it will have type $\alpha\ \mathsf{stream} \to \beta\mathsf{stream}$.

Component pre-conditions In order to be used in a sequential wrapper, the code portion should ensure that:

> no access to *free* variables are included in the code, that is no read (write) accesses to variables that are not included in the input (output) parameter list or declared internally are allowed, nor the definition of internal state through static variables is allowed.

We will refer to portions of code ensuring the property above as *pure functional* code (or *pure function*). It is worth pointing out that wrapper requires to explicitly state what's produced in output. A C++ function such as

```
vector<float> foo(vector<float> x, vector<float> y) {
  for(i : x) x[i]+=y[i];
  return(x);
}
```

does not provide the necessary information required by the wrapper component pre-condition, as it does not provide information about which vector belongs to the output parameter list[2].

Sample wrapper pattern The following C++ code snippet

```
for(int i=0; i<N; i++) {
    x[i] = x[i]*y[i];
  }
```

is code that may be legally wrapped in a sequential pattern provided `N, x[]` and `y[]` are named into the input parameter list and `x[]` is also named in the output parameter list
The function computed by the wrapper code will have type

$$\textsf{int} \times \textsf{vector<T>} \times \textsf{vector<T>} \rightarrow \textsf{vector<T>}$$

3.1.2 Sequential composition

Problem solved The pattern executes two computations using the same resources, sequentially.

Functional parameters The two computations to be serialized on the same resources (the pattern *stages*).

Non functional parameters The maximum parallelism degree to be used in case the two stages are parallel.

Implementation Implementation is trivial in case of sequential stages. In case the stages are parallel, a suitable set of resources should be used that may compute the first stage first and the second stage after the end of the computation of the first one.

[2]of course, the body code may be parsed and the information derived through simple data flow analysis, but it is not present in the function signature

Functional semantics If the first stage computes $f_1 : \alpha \rightarrow \beta$ and the second one computes $f_2 : \beta \rightarrow \gamma$, then their sequential composition will compute $fun(x) \rightarrow g(f(x)) : \alpha \rightarrow \gamma$. A sequential composition pattern acting as a parameter in a stream parallel pattern or composed to other stream parallel patterns will behave as a stream parallel patterns, that is it will have type α stream $\rightarrow \gamma$stream.

Parallel semantics All computation of the pattern happens sequentially, although both stages may be internally parallel.

Component pre-condition Both stages should be pure functions.

3.2 Stream parallel patterns

In this section we list the *stream parallel* patterns included in the initial **RePhrase** pattern set, that is those patterns exploiting 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 *map*, *filter*, *reduce* and *iterative* computations.

All the data parallel patterns process an input stream whose items have a type α (denoted as α stream) and produce a stream of output data items of type β (it may also be the case that $\alpha \equiv \beta$). It is worth pointing out that the number of items in a stream does not appear in the stream type.

3.2.1 Farm

Also known as This pattern is also referred to as *task farm* or *stream map*.

Problem solved 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.

Functional parameters The function to be computed is provided as sequential code or as a parameter “worker” pattern.

Non functional parameters Notable non functional parameters include:

- *parallelism degree*, the amount of concurrent activities to be performed in parallel when computing the pattern;
- *scheduling policy*, the policy to be used to distribute input tasks to the computational resources dedicated to compute in parallel the pattern function.

Implementation Implementation of the farm pattern will include a number of active entities (threads, processes)–usually called *workers*–all able to compute the farm function. The implementation may use active entities to schedule tasks to be computed to the active entities computing the farm function. Similarly, active entities can be used to gather results from the workers and transmit them onto the output stream. Otherwise, instead of active entities, workers can have access to passive data structures hosting the tasks to be computed and/or the result repository. The second solution is more suitable for coarse grain parallelism, while the first one requires additional concurrent activities for scheduling/gathering activities.

Functional semantics Assuming the type of the function computed by the workers is $f : \alpha \rightarrow \beta$, then the farm pattern will implement a function of type α stream $\rightarrow \beta$ stream. For each item x_i appearing on the input stream, an item $f(x_i)$ will be delivered onto the output stream.

Parallel semantics In principle, all the computation of $f(x_i)$ and of $f(x_j)\ \forall i, j : i \neq j$ may be computed in parallel. However, computation of $f(x_i)$ may start only at the moment x_i becomes available over the input stream. Therefore, assuming items appear on the input stream with an interarrival time T_a and the computation of f takes T_f, then at most $\frac{T_f}{T_A}$ computations will take place in parallel, at any time. This amount of computations potentially happening in parallel may be limited by the actual amount of computing resources available, which limit the maximum parallelism degree, i.e. number of worker entities.

Component pre-conditions The function $f : \alpha \rightarrow \beta$ computed by the workers on each stream element must be a pure function.

3.2.2 Pipeline

Problem solved 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.

Functional parameters The pipeline stages, expressed as functions or other stream parallel patterns.

Non functional parameters

Implementation A set of concurrent activities are used, each taking care of a single stage. An alternative solution is to use a farm whose workers implement, one after the other, sequentially, all the pipeline stages.

Functional semantics Provided the i-th stage in an n stage pipeline computes a function $f_i : \alpha \to \beta$, the pipeline computes a function of type α stream $\to$ β stream. For each stream item x an item $f_n(f_{n-1}(\ldots f_1(x)\ldots))$ is output delivered the pipeline output stream.

Parallel semantics Assuming that the items appearing on the input stream are $\ldots, x_{i+1}, x_i, x_{i-1}, \ldots$ the computation of stage f_j over the partial result relative to x_i will happen in parallel to the computation of f_{j+k} over the partial result relative to x_{i-k}.

Component pre-conditions All stages must implement pure functions

3.2.3 Stream filter pattern

Problem solved 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) $\mathcal{P} : \alpha \to \{true, false\}$. 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.

Functional parameters The boolean function (predicate) $\mathcal{P} : \alpha \to \{true, false\}$ to be used to decide whether or not a given input item is to be passed onto the output stream.

Non functional parameters Notable non functional parameters include parallelism degree and scheduling policies similar to the ones used in the farm pattern.

Implementation Depending on the time spent evaluating the filter function, a parallel structure similar to the one used to implement the farm may be used, with the worker entities passing or discarding their input values onto their output streams depending on the result given by the filter function computed onto the input data item.

Functional semantics The computation processes a sequence of input data items $\ldots, x_{i+1}, x_i, x_{i-1}, \ldots$ and produces a sequence of data items of the same type that may have a different cardinality. More specifically an input data item x_i is transmitted onto the output stream if and only if $\mathcal{P}(x_i)$ is true.

Parallel semantics In principle the filtering on different input data items x_i and x_j with $i \neq j$ may be evaluated in parallel. As in the farm pattern, the filtering function on each input data item x_i can be evaluated only at the moment x_i is received, i.e. become available over the input stream. Therefore, assuming the

items appear with an inter-arrival time T_A and the computation time of the filtering function is T_P then at most $\frac{T_P}{T_A}$ data items can be evaluated in parallel at any time. However, this is a theoretical limit, as the maximum number of items to be processed in parallel may be limited by the actual number of resources available.

Component pre-conditions The filtering function must have signature $\mathcal{P} : \alpha \to \{true, false\}$ and must be a pure function.

3.2.4 Stream accumulator pattern

Problem solved 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 $\oplus : \alpha \times \alpha \to \alpha$, although commutative and associative functions will provide much better and more scalable implementation.

Functional parameters The function $\oplus$ used to accumulate the values, the number of items $\mathcal{K}$ to be summed before actually delivering a results on the output stream (all, a given number, those arriving up to a certain condition is verified on the accumulator value) and the data structure (internal or external) to be used as accumulator.

Non functional parameters The parallelism degree, the kind of management for the concurrent accesses to the accumulator (mutually exclusive, associative, etc.).

Implementation The implementation of the pattern is mostly determined by the type of the "sum" function and by the kind of accumulator considered. At the worst case extreme, the updates may be computed in parallel but have to be committed serially. At the best case extreme, updates may be computed and applied concurrently.

Functional semantics The computation processes a sequence of input data items $\ldots, x_{i+1}, x_i, x_{i-1}, \ldots$ and produces a sequence of data items of the same type that may have a different cardinality. Each output data items corresponds to the application of the function $\oplus$ on a sequence of input data items that are collapsed into a single output value. The number of elements to be accumulated depends on the functional parameters provided by the operator. To exemplify a case, we mention a situation in which the first output data item y_1 of type α is obtained by accumulating the first burst (of $\mathcal{K}$ elements) of the input stream, the second output y_2 is applied on the second burst and so on according to the following relationship: $y_i = \sum_{j=1+(i-1)*\mathcal{K}}^{i*\mathcal{K}} x_j$ for any i.

Parallel semantics In the case of an associative and commutative function $\oplus$ the computation can be performed in parallel by several parallel entities (workers), each one applying the computation on a subset of the input elements and then their partial results are used to compute the final result of the accumulation. This can be calculated serially, by a single entity of the implementation, or in parallel according to a tree-based structure with proper interactions between workers. In the case of function $\oplus$ not associative and commutative, multiple threads/processes must synchronize and update the result of the computation atomically and in the proper order. In that case the speedup is limited since the updates to the data structure have to be performed serially.

Component pre-conditions The function $\oplus : \alpha \times \alpha \rightarrow \alpha$ must be a pure function.

3.2.5 Stream iteration pattern

Problem solved 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.

Functional parameters Notable functional parameters include the "nested" pattern to be iterated which must implement a function α stream $\rightarrow \alpha$ stream, a boolean predicate $\mathcal{P} : \alpha \rightarrow \{true, false\}$ stating whether the result of the "nested" pattern is to be flown again to the nested pattern input stream (through a logical "feedback" stream) or delivered onto the output stream.

Non functional parameters Notable non functional parameters include parallelism degree, scheduling policies relative to the delivery of pattern input and feedback stream data items to nested pattern inputs, e.g. a higher priority can be given to the feedback connection instead of the input stream or vice-versa, or the connection on which delivering input items to the inner pattern can be chosen according to a predicate operating on the internal state of the pattern.

Implementation The pattern may be implemented using dedicated concurrent activities to compute the feedback and the input guards or leaving these activities to be managed directly on the concurrent activities (the first and the last one) used to implement the nested pattern.

Functional semantics This patterns processes a sequence of input data items and for each data items apply a function $f : \alpha \rightarrow \alpha$ at least one time (but also several times) on each input element. In other words, each input data item y_i of type α is the result of the iterative computation (n times with $n \geq 1$) of the corresponding

input data item x_i, i.e. $y_i = f(f(...f(x_i)))$ with $n \geq 1$ application of the function f. The number of iterations n is the number of times the function f can be applied before the predicate $\mathcal{P}$ outcome is true.

Parallel semantics The parallel semantics of this pattern is the one of the inner pattern which is any stream-based parallel pattern of type α stream $\rightarrow \alpha$ stream. Therefore, if a farm pattern is used as the inner pattern, the application of f can happen in parallel on distinct input tuples. If f is a composition of function, the inner pattern can be the pipeline one. All the possible combinations and nesting are possible provided that the inner pattern respects the type of the input/output streams.

Component pre-conditions The function $f : \alpha \rightarrow \alpha$ must be a pure function as well as the predicate $\mathcal{P} : \alpha \rightarrow \{true, false\}$.

3.3 Data parallel patterns

In this section we list the *data parallel* patterns included in the initial pattern set, that is those patterns exploiting 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.

The data parallel patterns included in the initial pattern set include *map*, *reduce*, *stencil*, *divide and conquer* and *iterative* computations. All the data parallel patterns process an input collection whose items have a type α and produce either a collection of output data items of type β or a single output data of type β.

3.3.1 Map

Also known as: parallel for, apply-to-all.

Problem solved 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.

Functional parameters The computation (function f) to be applied on each data item of the collection to get the corresponding output collection item.

Non functional parameters Notable non functional parameter include the parallelism degree, the number of partitions to be generated out of the input collection for parallel processing (equal or greater than the parallelism degree), the scheduling policy of partitions to concurrent activity scheduling and the gathering policy of the results from the concurrent activities to be used to re-build the output collection.

Implementation A number of concurrent activities will be created, each capable of computing the function f over a partition of the input collection by producing the corresponding partition of the output collection. The computations should be synchronized, such that a correct output collection may be delivered as a result, that is all the corresponding partitions must be correctly gathered. In case the map pattern is used as functional parameter of a stream parallel pattern, than an iterator over the input stream items has to be prepended such that the map computation may process, in turn, all the input data items.

Functional semantics This patterns processes all the elements of an input collection and for each of these a function f is applied by producing the corresponding element of the output data collections. Therefore, the output is a collection $y_1, y_2, \ldots, y_N$ of data items of type β such that for each $i = 1, 2, \ldots, N$ $y_i = f(x_i)$ where x_i is the corresponding element of the input collection.

Parallel semantics All the elements of the input collection can be computed in parallel. In fact, in the map pattern, the computation on each data item of the input collection is independent from the other data items of the same collection. However, the maximum number of data items that can be computed in parallel depends actually on the parallelism degree of the pattern.

Component pre-conditions The function $f : \alpha \to \beta$ must be a pure function.

3.3.2 Stencil

Problem solved 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.

Functional parameters The computation $f : \alpha^* \to \alpha$ to be applied at each position of the input data structure, the kind of neighborhood to be considered (coordinates and number of neighbor elements) for the computation of the set of the values to be included in the output collection.

Non functional parameters Notable non functional parameters include the parallelism degree, the partitioning and collection policies to be used to produce the input data for (gather the results from) the stencil concurrent activities, similarly to the map pattern.

Implementation Implementations may be classified as "in place" or not. In an "in place" implementation, each sub-computation uses buffers to host the results that should be eventually committed in the output collection locations (the same hosting the initial values of the input collection when the computation is started). Commit of the buffers happens only when no other (sub-)computation needs the old values to be re-written to compute another neighborhood. In the not "in place" implementation the output collection is implemented by a different data structure than the original one. Multiple copies can be maintained if needed.

Functional semantics This pattern processes an input collection of data items of type α and produces an isomorphic collection of data items of the same type. The patterns executes for each input element a function $f : \alpha^* \rightarrow \alpha$ which produces the new value of that element. To compute it, the old value of the element and some of its neighbors are used to compute the function. The input of the function f are the old values of the elements of the input collection before being updated by the computation itself.

Parallel semantics All the elements of the input collection can be computed in parallel by a set of concurrent activities whose number (parallelism degree) limits the maximum number of data items that can be computed in parallel. Differently from the map patterns, concurrent activities are logically interdependent as to compute a data item a worker may need to access to elements logically assigned to other concurrent activities. Such interconnections may physically not exist in a concrete implementation, e.g. they may be resolved by the concurrent activities by scheduling portions of data (e.g. using overlapped partitions) or by sending all the elements needed to the computation of a data item to the owning worker.

Component pre-conditions The function $f : \alpha^* \rightarrow \alpha$ to be applied on each data item of the input collection must be a pure function.

3.3.3 Reduce

Problem solved The 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.

Functional parameters The computation modelling the "sum" function $\oplus$ and, possibly, the *neutral* value for $\oplus$ or some initial value of the accumulator.

Non functional parameters Notable non functional parameters include the parallelism degree, a partitioning algorithm to split the initial collection in chunks, a scheduling strategy to schedule the chunks to the available concurrent activities.

Implementation Depending on the properties of the $\oplus$ function, there are different implementation techniques suitable to implement the pattern. At the one extreme, local "sums" may be computed for all the partitions of the input data collection and a then the locally computed values may be either serially summed up into the final result or processed in parallel to provide the same result (e.g. in a tree way).

Functional semantics This patterns applied the operator $\oplus : \alpha \times \alpha \to \alpha$ on all the data items of the input collection. The result y of type α is obtained as follows: $y = x_1 \oplus x_2 \oplus \ldots x_N$ where x_i is the i-th data item of the input collection.

Parallel semantics According to the properties of the operator $\oplus$ the computation can be applied in parallel. The input collection can be partitioned in chunks (contiguous segments, interleaved ones, and so forth). A set of concurrent activities apply the operator $\oplus$ on each chunk independently by producing a set of N partial results where N is the parallelism degree. The computation of the final result can be calculated serially, by one of these concurrent activities, or in parallel according to tree-based structures.

Component pre-conditions The function $\oplus : \alpha \times \alpha \to \alpha$ to be applied on each data item of the input collection must be a pure function.

3.3.4 MapReduce

Know also as: Google mapreduce.

Problem solved 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 "sum" of the values output for the same key in the first "map" phase.

Functional parameters The computation to be applied to all the items of the input collection $f : \alpha \to \langle \text{Key}, \beta \rangle$ and the computation to be used to sum up the values relative to the same key ($\oplus : \beta \times \beta \to \beta$).

Non functional parameters Non functional parameters include the parallelism degree to be used for the MapReduce computation.

Implementation The implementation of MapReduce is largely dependent in the amount of data to be processed. Small input collections, producing a comparable size set of different keys, may be easily and efficiently computed *in memory* partitioning the input collection in subcollections, processing each subcollection in a different thread, producing a set of unique key value pairs per thread and eventually reducing the key value pairs produced by the different threads to the values to be included in the final result. When large collections have to be processed, usually the implementation moves the computation code to the places where the input collection partitions are stored for local processing.

Functional semantics The pattern applies the function f on all the items of an input data collection in a first step. In a second step all the partial results with the same key are summed together according to the binary function $\oplus$. The result of the pattern is a collection of data items of type β, one produced for each key.

Parallel semantics The pattern applies the function f on all the input data items in parallel. This parallelism is potentially bounded by the real parallelism degree of the computation. The second phase is a reduce on all the elements with the same key. This can also be computed in parallel by each concurrent activity on its partition before, and then the global result for each key can be computed serially by one of the concurrent activity or in parallel according to a tree-based structure.

Component pre-conditions The functions $f : \alpha \to \langle \text{Key}, \beta \rangle$ and $\oplus : \beta \times \beta \to \beta$ must be pure functions.

3.3.5 Divide and conquer

Problem solved 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.

Functional parameters The test for base case and the solution for the base case problem, the computation producing the sub-problems out of a non-base case problem and the computation building the result of a problem out of the solution of the sub-problems.

Non functional parameters Typical non functional parameters include the overall parallelism degree and the depth of the divide phase to be run before adopting a sequential implementation of the divide and conquer.

Implementation Several different implementations are possible, typically either exploiting the divide tree topology or using a set of concurrent activities able to divide, solve and conquer any intermediate problem/result. In the latter case, dynamic scheduling of problems and partial results to concurrent activities is needed.

Functional semantics The pattern applies a function $f : \alpha^* \to \beta^*$ on a collection of elements of types α by producing a collection of elements of the same of a different type. For the function f it is known a definition for input collections of a certain size. A decomposition function $\mathcal{S}$ is used to split the collection into distinct partitions up to the size of the base problem, which can be solved directly by using f. Finally the partial results of the base problems are combined together according to a merge function $\mathcal{M}$ in order to build the final output collection.

Parallel semantics The divide phase can be executed serially by a concurrent activity which distributes sub-problems to a set of parallel workers that compute them in parallel. The distribution logic is also in charge of combining results of smaller sub-problems into results of bigger ones produced by the workers, up to reaching the final output of the pattern. In the case the pattern is executed as a stream parallel patterns, the input is a stream of collection and the divide and conquer pattern is applied on each element of the stream (a collection) serially (each collection in parallel) or by mixing sub-problems of different stream elements for a better performance.

Component pre-conditions The functions f, the one to divide a problem $\mathcal{S}$ and to merge sub-problems into bigger ones $\mathcal{M}$ must be pure functions.

4. Data collection management

Stream and data parallel patterns process data coming from different places/sources. When a pattern is actually started, input data is assumed to be available according to different kind of semantics: items from a stream (available at successive times), partitions of a collection (available altogether), etc.

However, from time to time it is useful to be able to use patterns either producing or consuming data with no processing at all, according to stream or data parallelism semantics. Such patterns are particularly useful to provide initial data to or to retrieve results from parallel pattern compositions and to transform data to stream parallelism (or vice-versa) within a pattern composition.

We describe in the following sections two different classes of patterns: one producing/consuming data streams and one splitting/gathering data collections for data parallel processing.

4.1 Streams

Data management in streams provides just two patterns: a stream generator and a stream collapser. For the sake of simplicity, we assume that both patterns are *sequential*, although nothing prevents to consider the possibility to have parallel stream generators and collapsers.

4.1.1 Stream generator pattern

Problem solved Provide a stream of items to be processed by (a composition of) stream parallel patterns.

Functional parameters The computation producing the stream. We assume the computation sends data items onto the stream by invoking a kind of asynchronous `send`.

Non functional parameters The interdeparture time to be ensured.

Implementation Two kind of different computations may be used here: internal stream generator–basically an iterator over a (memory or disk based) data

structure–and external stream generator–basically an iterator receiving from outside world some data item to deliver `next`. The stream generator pattern may also assume the data structure to stream comes from some input stream as a single (collection) value.

Functional semantics The code implementing the generator computation is logically typed $\mathsf{some}(\alpha) \rightarrow \alpha\ \mathsf{stream}$. Ideally, the function may receive an input item from the input stream and generate a number of items onto the output stream or start producing items on the output stream just after being started[3].

Parallel semantics The implementation of the stream generator is sequential. No parallelism at all.

Component pre-conditions The code used to implement the stream generator function may be (usually is) stateful code. However, we assume it does not access any variable not defined as a parameter.

4.1.2 Stream collapser pattern

Problem solved The pattern receives data items from a stream and produces a single (collection) data structure out of them.

Functional parameters The computation accumulating stream items in a collection or in a variable.

Non functional parameters The interarrival time to be accepted/processed.

Implementation Data items collected from the input stream may be delivered to an external output stream or gathered into a data structure that may be eventually stored to memory or disk or delivered onto the pattern output stream.

Functional semantics The function computed by the pattern has type $\alpha\ \mathsf{stream} \rightarrow \beta$.

Parallel semantics The stream collapser runs completely sequentially.

Component pre-conditions As for the stream generator case, the function may be stateful but should not access any free variable.

[3] we adopt the $\mathsf{some}(\alpha)$ notation to indicate either a data of type α or nothing (the unit type)

4.2 Data collections

Collections may be partitioned into subcollections for further data parallel processing and sub-collections re-assembled in a single collection after parallel processing.

Differently from stream collapsers and generators, we assume both data collection partitioning and gathering may be implemented in parallel.

4.2.1 Splitter

Problem solved Split a collection into a set of possibly overlapping sub collections for further data parallel processing.

Functional parameters The policy to be used to split, the policy to be used to replicate data among sub partitions.

Non functional parameters The parallelism degree used to compute the sub collections.

Implementation In principle, the partition generation may be implemented in parallel. Tree structures parallel partitioning schemas may be implemented after recursive, divide&conquer style partitioning policies.

Functional semantics Data partitioning is logically implemented by applying a function from collection index space ($\mathcal{I}$) to sets of collection indexes: $p : \mathcal{I} \rightarrow \mathsf{set}(\mathcal{I})$. Then a number of partitions are computed as the members of the function set result. Each partition hosts the items from the input collection corresponding to the indexes in the set.

Parallel semantics Ideally, each partition can be computed independently once the result of the function p is known. It is worth pointing out that in case of shared memory architectures usually a partition *is* the set computed by p as the access to the corresponding items may be performed just using the collection indexes in the set.

Component pre-conditions None

4.2.2 Merger

Problem solved Merge a set of sub collections of data in a single collection after parallel processing.

Functional parameters The merge policy.

Non functional parameters The parallelism degree used to compute the merge.

Implementation The merger may be implemented in parallel (according to some recursive merge strategy) or sequentially. In case of shared memory target architectures, the merge operation may be implicit in the previous data parallel computation, as the concurrent activities that eventually produce the data items to be included in the merged collection may produce them directly in the correct locations.

Functional semantics Usually, data merge corresponds to the placement of each one of the sub collection items into a place of the resulting collection corresponding to the place where the subcollection come from in the split pattern that originated the parallel computation. In case the result collection is different, the merge policy should specify an index in the resulting collection for each one of the items in the subcollections. Ideally this may be provided by a function from sub collection, item index to result collection index.

Parallel semantics In case the merge is implemented in parallel the parallel semantics is derived from the parallel semantics of the inner patterns.

Component pre-conditions None.

5. Stateful patterns

The list of patterns in Sec. 3.2 and 3.3 only include patterns with stateless components. Pipeline stages and farm workers are assumed to be stateless as the mapreduce or BSP workers. However, a number of computations happen to need some notion of stateful computation.

For any of the patterns listed above, we assumed to have stateful variants where the different concurrent activities involved in pattern execution maintain and process local or global state variables.

As an example a stateful farm my have portions of a global state allocated internally to the concurrent activities used to process the different input stream data items and it may run particular scheduling policies directing input tasks to workers in charge of managing particular sub-portions of the global state.

Within the initial set of **RePhrase** patterns, we assume each of the patterns listed in the previous sections may have a further functional parameter stating:

1. the type of the global state
2. possibly, a partitioning schema of the global state among the parallel activities modelled by the patterns
3. possibly, a characterization of the kind of actions performed onto the state (components).

such that parallel activities in the pattern may access the state accordingly within their "body" activities. The implementation of the pattern will ensure the state semantics is actually implemented.

We assume (for the initial pattern set) the existence of the following "state partition schemas":

Centralized the state data structure is unique and centralized (w.r.t. the parallel activities in the pattern). All concurrent activities may access the state in read and write mode, but this requires proper synchronization.

Partitioned the state data structure is partitioned among all the concurrent activities in the pattern. All concurrent activities may access any one of the partitions in read or write access mode. Proper synchronization is needed as in the centralized case.

Locale the state data structure is partitioned among all the concurrent activities in the pattern. Each concurrent activity may read and write just the local partition of the state. No synchronization is required, in principle, unless the concurrent activity is itself parallel (as of some pattern composition).

Single owner the state data structure is partitioned among all the the concurrent activities of the pattern. A concurrent activity may read any of the state partitions but it can only write the local partition. Proper synchronization mechanisms are required, consequently.

We also assume that the operations performed on the state (components) by the concurrent activities in the pattern[4] may be classified as follows:

Resource no knowledge available relative the access pattern, but for the fact the pattern accesses performs arbitrary both read and write operations

Readonly the operation just reads the state data structures

Accumulator the same associate and commutative operation is performed by all the operation on the state.

In **RePhrase** we assume no other constrain/semantics but those deriving from the different kind of partitioning schemas and access functions listed above. As an example, let us consider a farm pattern with a centralized state of type α where all the workers access the state by means of a "resource" function. The farm implementation must ensure that the accesses performed from the workers are managed in mutual exclusion. This is the only thing to be ensured. As an example, a user wishing to grant priorities in the accesses to the shared state should provide his own priority access state implementation policies.

[4] all concurrent activities use the same kind of operation on a single state data structure

6. Pattern composition rules

Usually, parallel patterns may be nested more or less arbitrary during parallel application design. Within **RePhrase**, we assume that the patterns in the **RePhrase** initial pattern set may only be nested according the following rule:

- stream parallel patterns may be parameters of stream parallel patterns
- data parallel patterns may be parameters of data parallel patterns
- data parallel patterns may be parameters of stream parallel patterns
- data parallel patterns in stream parallel computations should implement data parallel computation on each one of the stream items of the input stream, processing all the streams appearing onto the input stream one after the other
- stream parallel pattern composition should always start with a stream generator and end with a stream collapser
- data parallel pattern compositions should start with a splitter and end with a merger
- any kind of patterns may have sequential code wrapper internal parameters

This is needed to ensure correctness, efficiency and to implement the proven "two-tier" schema that has already been proven suitable in the algorithmic skeleton framework [5, 10]. According to the two tier composition schema, in a pattern (algorithmic skeleton nesting) stream parallel patterns (if any) occupy the top positions of the pattern nesting tree, sequential patterns are placed into the leaves and data parallel pattern may be placed in intermediate nodes, in between stream parallel and sequential patterns.

7. RePhrase pattern DSL

In order to be able to support conscious and efficient design of patterned parallel applications. We propose to implement an external domain specific language suitable to support exploration of the space of patterned implementations of the very same application.

The space of implementation is defined by the functional equivalence preserving rewritings available for pattern compositions (see Sec.7.3 below).

Each item in the space is characterized by:

- having the same functional semantics, that is computing the same results out of the same inputs, and
- having different values for the non functional properties of interest, that is different performance values (completion or service time), efficiency, energy consumed per FLOP/per run, security properties, etc.

The **RePhrase** pattern DSL will therefore provide facilities to

- express patterns and pattern compositions, along with their functional parameters,
- annotate pattern expression with non functional property values
- rewrite pattern expressions into functionally equivalent pattern expressions through the application of know and verified rewriting rules
- evaluate (ideal) non functional property values in pattern expressions through the application of different kind of (simplified) non functional property models, and
- transform pattern expressions into pattern expressions according to well known and efficient optimization heuristics/algorithms
- to generate template patterned code ready to be completed with business logic code (sequential code fragments and actual data type) for different hw/sw target architectures.

7.1 High level pattern language

The **RePhrase** pattern DSL is an abstract grammar whose terms represent either sequential code patterns (sequential functions, stream generators and collapsers) or parallel patterns (stream parallel, data parallel, Splitters and Mergers.

The grammar is defined in such a way each pattern is represented with a functor and a set of parameters modelling its functional parameters.

The set of patterns described in the previous Sections is captured by the following grammar:

```
SeqPattern ::=SeqWrapper | SeqComp |
              StreamGenerator(Policy) | StreamCollapser(Policy)
SPpattern  ::= Farm(Pattern) | Pipe(Pattern,Pattern) |
              ...
DPpattern  ::= Map(Pattern) | Reduce(Pattern) | MapReduce(Pattern,Pattern) |
              ...
Pattern    ::= SeqPattern | SPpattern | DPpattern | Splitter | Merger
```

This is an abstract grammar. The actual implementation of the DSL may possibly provide slightly different terminal symbols, while preserving the structure of this abstract grammar.

7.2 Annotations

Each term in the pattern grammar may be annotated with some non functional property value. The annotations will be therefore defined as

```
Annotation ::= Functor(Value) | Annotation list
Functor    ::= NonFunctionalPropertyName
NonFunctionalPropertyName ::=
              ServiceTime | CompletionTime | Latency
              ParDegree |
              SecureCode | SecureData | SecureBoth |
              ...
```

Annotations may be:

- user provided (manual edit)
- system provided (e.g. through automated profiling)
- system derived (e.g. applying non functional property models to the available annotation values)

Rule name	Rule
Farm intro/elim	SeqWrapper() $\equiv$ Farm(SeqWrapper())
Pipe intro/elim	Pipe(Pat$_a$, Pat$_b$) $\equiv$ SeqComp(Pat$_a$, Pat$_b$)
Map fusion/promotion	Map(SeqComp(Pat$_a$, Pat$_b$)) $\equiv$ SeqComp(Map(Pat$_a$),Map(Pat$_b$))
Reduce localization	Pipe(Map(Pat$_a$),Reduce(Pat$_b$)) $\equiv$ Pipe(SeqComp(Map(Pat$_a$),Reduce(Pat$_b$)),Reduce(Pat$_b$))
D2S par	Map(Pat) $\rightarrow$ Pipe(StreamSource(), Farm(Pat), StreamCollapse())

Table 7.2: Stateless pattern rewriting rules

As an example, the application programmer may provide through annotations the required parallelism degree of a pattern (by associating the annotation ServiceTime(n) to the pattern), the sequential portions of business code may be decorated with the automatically profiled latency (associating annotations Latency(v-msec) to the SeqWrapper patterns), or the latency of a Pipeline pattern may be derived from the latency of the pipeline components (the pattern stages) exploiting the abstract performance model associated to the pipeline pattern stating that

$$Lat(pipe) = \sum_{i \in stages} Lat(stage_i)$$

7.3 Rewriting rules

A library of pattern expression rewriting rules is provided with the DSL, hosting all the known and useful pattern rewriting rules.

The library will include well known rule as the ones listed in Table 7.2 valid for stateless patterns as well as more sophisticated rules also taking into account stateful patterns.

As an example, a stateful pattern rewriting rule may state that a SeqWrapper pattern with an "accumulator" centralized state may be rewritten into a Farm provided all the farm workers share a "centralized accumulator reference". In case the centralized state of the SeqWrapper is a "generic" state, additional constrains may be required to apply the same rule while preserving the computation functional semantics. In this case the rewriting rule will be written as a precondition/action rule such as:

$$\frac{\mathsf{SeqWrapper}(\ldots)\,\textbf{with}\ \mathsf{State(int,\ centralized,\ accumulator)}}{\mathsf{Farm}(\mathsf{SeqWrapper}(\ldots))\,\textbf{with}\mathsf{State}(\mathsf{int}, \mathsf{centralized}, \mathsf{accumulator})}\,\textbf{Farm intro rule}$$

A more complete set of rewriting rules will be provided in the report describing the first implementation of the initial pattern set D2.4 at project M16.

7.4 (Non functional) property models

The **RePhrase** pattern DSL has associated a number of non functional property models, that is algorithms, protocols, heuristics suitable to associate to an annotated pattern expression in the DSL a value for a given non functional property.

We assume to have at least models for

- performance (completion time, service time), capable of computing the performance measures of interest when a small set of annotations relative to the measures achieved in the base components (e.g. the sequential code wrappers) are known, and

- parallelism degree, capable of figuring out the ideal parallelism degree of the application, once parallelism degrees of basic components are present in base component pattern expression annotations.

Among the different kind of models available for the non functional properties of interest, we will consider the possibility to use:

- analytical (approximated) models (e.g. the ones for performance)

- learning based models (e.g. neural networks)

- heuristic based models (e.g. rule of thumb style models)

In any case the models should be stored in the DSL library as functions mapping annotated pattern expressions to annotated pattern expressions. The library itself should be implemented in such a way it can be easily extended provided more and more precise models as soon as they become available.

7.5 Optimizers

Last but not least, some well known optimization strategies exists for particular class of parallel patterns (see map fusion for data parallel patterns (pattern expressions) [6] and normal form for stream parallel patterns (pattern expressions) [1]). Within the DSL framework, suitable "optimizers" will be provided implementing these optimization strategies as functions from pattern expressions to pattern expressions. These functions could be invoked by the parallel application programmer running the the design space exploration process in two different cases:

- on the pattern expression representing the whole parallel structure of the application under design. In this case the application programmer simply trust the positive effects of the optimization strategy invoked and leaves the DSL framework the full duty relative to the parallel code optimization.

- on a component of the pattern expression representing part of the whole parallel structure of the application under design. In this case the application programmer relies on known optimizations only for part of the application parallel structure and he will optimize the design of the whole application applying the full **RePhrase** design methodology to the overall pattern expression modelling the parallelism of the application.

7.6 DSL implementation

At the moment begin (M6 from **RePhrase** start), a small, proof-of-concept and limited version of a DSL design framework such as the one outlined in the previous sections has been developed. The prototype is written in Ocaml and only supports stateless stream parallel skeletons along with some classical functional semantics preserving rewriting rules. The code also provides simple modelling of service times (performance) and parallelism degree (resource usage). In addition, simple optimizers for pipeline service time and stream parallel normal form are provided.

Full implementation of the **RePhrase** DSL–targeting only the initial pattern set as described in this deliverable–will be provided in D2.4 at M16 and may possibly be completely different (in terms of implementation language and feature) completely different from the proof-of-concept prototype.

Also, the proof-of-concept prototype will not be released as open source but just used within the project to experiment and prepare the first version of the DSL framework to be delivered with D2.4.

8. Conclusion

This deliverable introduces the initial parallel design pattern set that will be used within **RePhrase** to introduce parallelism into existing applications or in applications designed from scratch.

The initial parallel pattern set will be used to start considering possible parallelizations of **RePhrase** use cases, as well as to experiment all the software engineering techniques that will be eventually provided in the **RePhrase** programming framework.

The pattern set contains well known parallel patterns suitable to represent the most common parallelism exploitation situations in the data intensive applications taken into account in **RePhrase**.

It will be expanded with new patterns, if the case, suitable to model particular situations managed in the **RePhrase** use cases or recognized in the international community as patterns worth to be considered new, alternative (w.r.t. the one already included in the initial pattern set) general purpose parallel patterns.

The final part of the deliverable outlines the main features that will be included in the **RePhrase** parallel pattern DSL. This DSL will be provided to the application programmers as a framework suitable to support experiments related to the design of the application parallel structure. The DSL will be completely defined and a first implementation will be provided in D2.4 (due at M16 in 10 months since now) along with the implementation of the initial pattern set components in the different parallel programming frameworks considered in the project. At that time the application programmer will be enabled to use the DSL to evaluate alternative parallel pattern compositions modelling their application and eventually to get actual application code from refactoring and code generation tools taking the final DSL pattern composition developed by the programmer as the final application parallel structure.

Bibliography

[1] Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *in proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems, MIT*, pages 955–962. IASTED/ACTA press, 1999.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[3] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, March 2004.

[4] M. Danelutto. On skeletons and design patterns. In *Proceedings of the International Conference PARCO 2001*, 2001. Imperial College Press.

[5] Marco Danelutto, Fabrizio Pasqualetti, and Susanna Pelagatti. Skeletons for data parallelism in p3l. In *Euro-Par '97 Parallel Processing, Third International Euro-Par Conference, Passau, Germany, August 26-29, 1997, Proceedings*, volume 1300 of *Lecture Notes in Computer Science*, pages 619–628. Springer, 1997.

[6] Kento Emoto and Kiminori Matsuzaki. An automatic fusion mechanism for variable-length list skeletons in sketo. *Int. J. Parallel Program.*, 42(4):546–563, August 2014.

[7] 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.

[8] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In *Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011,*

Turin, Italy, October 3-5, 2011, Revised Selected Papers, volume 7542 of *Lecture Notes in Computer Science*, pages 218–236. Springer, 2011.

[9] Kurt Keutzer and Tim Mattson. Introduction to Design Patterns for Parallel Computing. In David Patterson, Dennis Gannon, and Michael Wrinn, editors, *The Berkeley Par Lab: Progress in the Parallel Computing Landscape*. 2013. Chapter 8, available at `http://research.microsoft.com/en-us/um/redmond/about/collaboration/Par-lab-book/Chapter-8.pdf`.

[10] Herbert Kuchen. A skeleton library. In *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany, August 27-30, 2002, Proceedings*, volume 2400 of *Lecture Notes in Computer Science*, pages 620–629. Springer, 2002.

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

[12] 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.

[13] Jorge Luis Ortega-Arjona. *Patterns for Parallel Software Design*. Wiley, 2010. ISBN: 978-0-470-69734-4.