Final Pattern Definition Report
D2.5

Due date of deliverable: July 31\textsuperscript{st}, 2013
Actual submission date: July 31\textsuperscript{st}, 2013

\textit{Start date of project: 1\textsuperscript{st} October 2011}
\textit{Type: Deliverable}
\textit{WP number: WP2}
\textit{Task number: T2.1}

\textit{Responsible institution: UNIPI}
\textit{Editor & and editor’s address: Sonia Campa and Massimo Torquati, UNIPI}

\textbf{Dissemination Level}
\begin{center}
\begin{tabular}{|c|p{13cm}|}
\hline
PU & Public & \textbf{X} \\
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) \\
\hline
\end{tabular}
\end{center}
Executive summary

The deliverable reports on the parallel patterns identified by the project beneficiaries as useful to support the development of generic parallel application as well as of the specific use cases identified by the beneficiaries contributing to WP6.

The identified parallel patterns are described in two different parts of the document. The patterns listed in Chap. 2 are the ones needed to implement the WP6 use cases or the ones already demonstrated useful in a wide range of parallel applications. We expect these patterns will all be provided—by the end of the project—as suitable (composition of) skeletons implemented in C++/FastFlow and/or in Erlang, according to the framework(s) used to implement the use cases. We expect as well that for these patterns WP4 will provide suitable refactoring support and WP3 will provide appropriate mapping/scheduling policies. The patterns in Chap. 3 instead, are parallel patterns that have anyway been identified as useful ones, but that will not be implemented within the ParaPhrase project, as the implementation work will concentrate on more basic patterns and on those patterns actually needed to support the use case implementation.

The description of the patterns within the deliverable is high level and given accordingly to the style usually adopted when describing design patterns: the pattern (1), the functional and non functional parameters needed to instantiate the pattern (2) and a short description of the associated parallelism exploitation strategies and policies (3) are given for each one of the patterns. The pattern description is intentionally concise such that on the one hand the key aspects of the pattern are all evidenced and on the other hand we do not engage describing the implementation details that will eventually be influenced by a number of technical aspects that may be correctly taken into account only when the actual implementation of these patterns will be considered.
Figure 1: Positioning of Deliverable D2.5: w.r.t. other WP2 deliverables (top) and w.r.t. the forthcoming deliverables from other WPs (bottom).
Contents

1 Introduction 4

2 The Paraphrase pattern set 6
   2.1 Generic pattern set - core patterns 6
      2.1.1 Sequential pattern (wrapper) 6
      2.1.2 Pipeline pattern 7
      2.1.3 Farm pattern 7
      2.1.4 Map pattern 8
      2.1.5 Reduce pattern 9
   2.2 Generic pattern set - high level patterns 11
      2.2.1 Divide and conquer 11
      2.2.2 Search 12
      2.2.3 Sort 13
      2.2.4 Pool evolution pattern 13
      2.2.5 Work-flow graph interpreter 14
   2.3 Domain specific pattern set 16
      2.3.1 Orbit pattern 16
      2.3.2 (Colored) Stencil Pattern 17
      2.3.3 Instances of work-flow graph interpreter pattern 18
      2.3.4 Instances of the pool evolution pattern 18

3 Other patterns 22
   3.1 MapReduce 22
   3.2 Generalized zip 23
   3.3 Scan 24
   3.4 Fork/Join 24
   3.5 Iterated Map with Dependency Tracking 25
   3.6 Branch&Bound 26
Chapter 1

Introduction

In this deliverable we classified parallel patterns in Paraphrase pattern set and Other patterns. The goal is to provide through the former group an overview of the pattern set that will be implemented by the end of the project and will be used by the partners working on the applicative use cases; the latter describes a group of pattern that will not be implemented since they do not strictly fit the use cases requirements, but that we consider worth to discuss because they enforce the efficacy of our approach.

As stated in [17] and [18], the Paraphrase pattern set classifies patterns between generic and domain specific. The generic set exploits a group of RISC patterns, very general, highly composable form of parallelism representing the basic building block of every parallel application. The domain specific set exploits more complex patterns which may be described in terms of generic ones and that define recurrent patterns of parallelism in specific applicative fields (for example, linear algebra, soft computing, genetic programming).

Following the methodological guideline defined in [17], a pattern is an abstraction defining recurrent parallelism exploitation schemas. Patterns do not focus on the actual software implementation nor on the hardware architecture at hand. They specify a solution to the recurrent problem of shaping the parallel behavior of an application. In most cases, one or more patterns, suitably nested/composed may be used to shape the parallel behaviour of a given applications. Due to descriptive role typical of parallel design patterns, the pattern set is described in this document in a synthetic “plain English text” way, as it typically happens in the design pattern software engineering community. Each pattern description includes a Description paragraph that defines the main pattern features, a Parameter paragraph that enumerates the parameter(s) needed to instantiate the given pattern, and eventually an Implementation section that provides an overview of the possible implementations of the pattern in terms of graph/tree of computational nodes and concurrent activities. Note that this description is related to which parameters have to provided in order to instantiate a pattern; then, one or more arguments could be needed to actually make it running. In particular, input/output arguments (such as streams of
item or collection of data) have to be further specified in an actual implementation.

We recall that the Paraphrase methodology envisions a three layer programming approach. At higher level, the application is a composition of parallel patterns, such as the ones described in this document. At a second, intermediate level, the patterns are implemented in terms of skeletons, that is higher order functions or template classes implementing and encapsulating suitable parallel activities graphs that efficiently implement the high level parallel pattern structure. The skeletons used to actually implement the patterns used by the application programmer to express the parallel behaviour of the business code are developed, tuned and optimized by system programmers, rather than by the application programmers. At a third level, the skeletons eventually used to implement the high level parallel patterns are implemented as C++/Fastflow programs or as Erlang programs using all the programming tools and facilities available on the target heterogeneous architecture, possibly encapsulating and hiding all those programming details related to communication, data access and heterogeneity of the architecture at hand.

By the end of the project, patterns defined in Section 2 will be implemented as C++ skeleton templates and/or in Erlang skeletons. The more general patterns, such as those listed in Sec. 2.1, will be implemented both in the C++/FastFlow and in the Erlang frameworks. The high level and the domain specific patterns listed in Sec. 2.2 and 2.3 will be only implemented in the framework(s) used to implement the use cases exploiting those particular patterns.

All the patterns listed in this report are considered patterns accepting just one input and producing one output (stream), to ensure full compositionality and therefore to enforce the structured parallel programming model adopted within the Paraphrase project. In particular, the full compositionality ensured by the single input/single output pattern model guarantees the possibility to express very complex parallel structure of applications through proper composition of core, high level and domain specific patterns.

Last but not least, both generic and domain specific pattern sets include patterns for describing data as well as task parallelism.
Chapter 2

The Paraphrase pattern set

This Chapter hosts the description of the patterns that will be eventually implemented in the project. Sec. 2.1 hosts the description of the RISC core patterns. Sec. 2.2 hosts the description of slightly higher level\(^1\) patterns. Eventually, Sec 2.3 hosts the description of different domain specific patterns whose usage has been envisaged in different applications from the Use case ParaPhrase work package.

2.1 Generic pattern set - core patterns

2.1.1 Sequential pattern (wrapper)

**Description** The sequential pattern is used to wrap sequential portions of code in such a way they can be used as parameters of other patterns. The sequential pattern is in fact the minimal building block needed in order to build (composition of) patterns. The sequential pattern models an independent computational entity accepting tasks as inputs and sequentially producing results as outputs.

**Parameters** The sequential pattern parameters include: the input task type \(T_1\), the output result type \(T_2\), and (a reference to) the sequential function to be wrapped. A conveniently instantiated sequential pattern will take as argument an item of type \(T_1\) as input and will produce an output of type \(T_2\) as output. The items could belong to a stream of items.

**Implementation** This pattern is generally implemented as a computational node built by wrapping the user-defined sequential code. The node runs sequentially over the target architecture and, as part of a composition of patterns, it could be connected to other computational nodes linked to its input or output channel.

\(^1\) of abstraction
2.1.2 Pipeline pattern

**Description**  The pipeline pattern models stream parallel computations organized as a sequence of stages. Given a sequence of stages $S_1, \ldots, S_k$, such that stage $S_i$ computes a function $f_i$ on the inputs coming from the computation of the previous stage $S_{i-1}$ and delivers the results to the next stage $S_{i+1}$, the pipeline pattern computes $f_k(f_{k-1}(\ldots(f_2(f_1(x_i))))\ldots))$ for each one of the items $x_j$ appearing onto the input stream. The computation of the different stages happens in parallel. Therefore, at any time, the computations of

$$f_1(x_t) \ f_2(f_1(x_{t-1})) \ \ldots \ f_n(f_{k-1}(\ldots(f_2(f_1(x_{t-k}))))\ldots))$$

all take place in parallel.

**Parameters**  A pipeline pattern takes as functional parameters the (references to a) set of patterns representing the internal behavior of the stages. A conveniently instantiated pipeline pattern will take as argument a stream of independent items as input and will produce a stream of independent items as output.

**Implementation**  The typical implementation of the pattern is provided as a set of parallel stages (nodes, classes, templates, etc.) which are connected in row so that, from the first to the latter, they communicate one-to-one with one receiver and one sender. The first stage could communicate with the input stream (or generates the stream) and the latter with the output stream (or it consumes the stream). Usually, the pipeline is provided as a template or a library call the user can instance in order to include it in the program design [10, 14, 16].

2.1.3 Farm pattern

**Description**  The farm pattern models an embarrassingly stream parallel computation in which a computing element called worker is replicated in a number of copies and each instance computes a different element of an input stream of items. Items to workers are dispatched and gathered through different policies, thus providing different behaviors to the same pattern from a computational point of view. Therefore, given an input stream $\langle x_1, \ldots, x_m \rangle$ and assuming the function computed by the workers is $f_w$, the farm pattern computes

$$f_w(x_1), \ldots, f_w(x_m)$$

and the computations of any $f_w(x_i)$ and $f_w(x_j)$ ($i \neq j$) may in principle be performed in parallel.

**Parameters**  The farm pattern functional parameters include the a reference to the pattern implementing the worker. The non functional parameters include the
number of workers, i.e. the parallelism degree of the farm, as well as the scheduling and gathering policies to be used to deploy tasks from the input stream and to gather results computed by the workers to the output stream. The possibility to require the constraint that the relative ordering of tasks and results is maintained is also denoted via a proper non functional parameter. Although the number of workers in a farm may in principle be equal to the number of items appearing onto the input stream, the available amount of processing resources as well as the overheads introduced in the scheduling and gathering phases may limit the actual number of workers.

A conveniently instantiated farm pattern will take as argument a stream of independent items as input and will produce a stream of independent items as output.

Implementation Traditionally, the farm pattern is implemented as a set of computational nodes [14,16], plus two additional nodes called emitter and collector [9]. The emitter is responsible for dispatching the input items to the workers, while the collector gathers all the results and delivers them to the farm pattern output stream. Both emitter and collector can be provided with one or more (parametric) policies in order to define how to schedule tasks and recollect results. The fact that the farm is described as a set of logically different nodes, does not imply that each node will be allocated on a physical resource. How nodes are distributed on the actual architecture can depend on (and influences) possible performance goals.

2.1.4 Map pattern

Description The map pattern models embarrassingly data parallel computations. In particular, the map pattern models those computations computing a new collection B (with generic element $b_i$) out of a collection A (with generic element $a_i$) by computing the generic element of B as a function of the generic element of collection A: $b_i = f(a_i)$. All the computations $b_i = f(a_i)$ are logically performed in parallel. The only difference with the farm pattern described in Sec. 2.1.3 is therefore relative to the source of the data items processed in parallel: here they come from the decomposition of a data collection (a vector, an array, an hash map, etc.) while in the farm pattern they come from an input stream and they are not available all at the same time.

Parameters The functional parameters of the map pattern include the input collection (and therefore its type and size) and a reference to the pattern representing the parallel function to be applied on the collection items. Non functional parameters include the parallelism degree. In principle, the parallelism degree may be the same as the size of the input collection, i.e. the parallelism grain is expressed at collection’s item level. However, in general these measures could be not related and a smaller parallelism degree is used which provides a boundary for the number of chunks in which the collection has to be partitioned in order to evaluated it in parallel [16].
Implementation  Data parallel patterns have been implemented both at library levels (as skeleton templates, as in FastFlow, for instance), and at data level providing libraries of parallel data on which the pattern is provided as a possible operation on data [12, 13, 15]. Generally speaking, the map pattern is embarrassingly parallel and the given function is applied in parallel to each item of the input collection; in some cases (because of the availability of resources, the algorithm, performance constraints, etc.), the input collection can be partitioned and each partition can be evaluated in parallel.

A conveniently instantiated map pattern will take as argument a collection of items as input and will produce a collection of items as output. If the map represents a truly embarrassingly parallel computation, input and output collections share the same sizes.

2.1.5 Reduce pattern

Description  The reduce pattern models tree structured data parallel computations where an input collection is processed to compute a scalar result by “summing up” all the collection items by means of an associative and commutative operator \( \oplus \). Therefore, given an input collection \([x_1, \ldots, x_m]\) the result computed by the reduce pattern is

\[
x_1 \oplus \ldots \oplus x_m
\]

The function \( \oplus \) is commutative and associative, thus it is feasible to apply it in parallel over a set of partitions of items and then iteratively proceed in the reduction phase over the partial results until a single value is reached.

Parameters  The reduce pattern functional parameters include (a reference to) the \( \oplus \) reduction function. The non functional parameters include the parallelism degree to be used while computing the reduce pattern.

As a member of the data parallel patterns family, a conveniently instantiated reduce pattern will take as argument a collection of items as input and will produce a single item (the result of the reduction) as output.

Implementation  As in the case of the map pattern, a pure data parallel implementation of this pattern would require to apply the reduction function to every non-overlapping couple of items, thus taking the single item as the grain for parallelism. However, depending on the available resources and/or the type of the reduction function, each step of reduction could also process a partition of items, thus making the parallelism degree (determining the size of the partitions) a mandatory parameter.

The reduce pattern is typically implemented as a set of computational nodes each computing the reduce \( \oplus \) independently over a partition of the input collection items. Partial results are then collected and the reducing \( \oplus \) phase is iterated until a single value is produced. As in the previous patterns, implementations can be
provided as library calls [2], skeleton templates [9], or methods implemented over parallel data structures [6].
2.2 Generic pattern set - high level patterns

2.2.1 Divide and conquer

Description The divide and conquer pattern is very common in many scientific applications where a given, complex problem can be solved by combining two different phase: a divide and a conquer phase. In the first phase, the divide one, the problem is recursively decomposed into simpler sub-problems until a given condition is reached. Generally, such condition is the one recognising if the sub-problem can be immediately solved (it represents the so called base case for which a solution is known) thus producing a sub-solution. The second phase, the conquer phase, recursively combines the partial sub-solutions computed during the divide phase up to the point the solution of the original problem is computed. Therefore, the Divide and Conquer pattern builds a k-tree of nodes (where k represents the number of sub-problems obtained at each split step) during the divide phase. The set of leafs of this tree represents the set of base cases and therefore the set of base sub-case solutions. In the conquer phase, the tree is visited backwards applying the conquer function to each node sub-solutions, up to the point the root node solution is computed.

More formally, given an input data set \( X \), a divide function \( d \) computing a set of subsets of \( X \), a boolean function evaluating whether a data set represents a base case \( b \), a function computing the solution of a base case \( s \) and function building the solution of a non base case by combining the solutions of its sub-cases \( c \), the result computed by a divide and conquer pattern may be represented by the recursive function:

1: \[\text{function} \ \text{DIVIDE} & \text{CONQUER}(\text{div}, \text{basecase}, \text{solve}, \text{conq})\]
2: \[\text{if} \ \text{basecase}(X) \ \text{then}\]
3: \[\text{return} \ \text{solve}(X)\]
4: \[\text{else}\]
5: \[\text{return} \ \text{conq}(\text{map}(\text{divide} & \text{conquer}(\text{div}, \text{basecase}, \text{solve}, \text{conq}))(\text{div}(X)))\]
6: \[\text{end if}\]
7: \[\text{end function}\]

Parameters The divide and conquer pattern functional parameters include the (references to the) functions checking the base case condition, computing the base case solution, dividing a problem into sub-problems and building the problem solution by combining the sub-problem solutions. The pattern non functional parameters include the parallelism degree and, possibly, a parameter defining the dimension of the (sub-)problems to be considered base cases. This latter parameter may be used to raise the computational grain of the base case solutions and therefore to improve the efficiency of the divide and conquer implementation. The divide and conquer pattern will take a representation of the problem as input (for instance a collection of data items) and will produce a solution (i.e. a single item or a collection as well) as output.
Implementation This pattern has been implemented in a number of skeleton libraries ([6, 7, 9, 11]). It is well represented by a tree of nodes that dynamically grows in depth until the “dividing” phase is proceeding. As soon as the base condition is reached at one node’s level, the “merging” phase starts, often activating a backward traversal of the tree. Thus, each node tests the base case condition and decides if producing tasks as a set of sub-problems, otherwise they evaluate the base case. In order to match the available processing resources, the traversing of the divide and conquer tree is often implemented by a) representing the tree in a suitable, shared, “task pool” data structure and b) establishing a pool of “worker” activities fetching nodes from the task pool and computing either the divide or the conquer phase relative to the extracted node.

2.2.2 Search

Description The search pattern looks for items in a collection \( A \) (a graph, a tree, an array, etc.) for those a given target boolean function \( f \) returns true. As an example, the function \( f \) may be defined as \( x \) being equal to a particular value \( x_{\text{target}} \). Two variants of the pattern will be considered: in the first one, the pattern visits in parallel the input collection \( A \) gathering all the elements \( a_i \) in the collection such that \( f(a_i) = \text{true} \) into an output “found items” result collection \( R \). In the second variant, the pattern just looks for the first item \( x \) such that \( f(x) = \text{true} \) and returns it as soon it is found.

Parameters The search pattern functional parameters include the (reference to the) target function used to evaluate whether an input collection item has to be inserted into the “found” item collection (the search pattern result), and possibly whether to stop searching or to continue and a parameter specifying if a set of found items or just the first found item must be returned. The non functional parameters include the parallelism degree to be used while visiting the input collection.

In case a divide and conquer pattern is used to visit the input collection, all the functional and non functional parameters of the divide and conquer pattern must be taken into account as parameters of the search pattern.

Generally speaking, a conveniently instanced search pattern will require a collection of items as input argument and produce one item (or none if the search fails) as output.

Implementation In case the input collection is a recursive data structure (e.g. a tree), the patterns visits (breath-first) the collection by means of a recursive parallel visit algorithm modelled by a divide and conquer pattern after the recursive definition of the collection data type. In case the input collection is a classic “array” (or vector) data collection, the visit is performed using a map data parallel pattern.

The Mallba project [1] provides this pattern in the form of a set of parametric skeleton implementing different search strategies.
2.2.3 Sort

**Description** The sorting pattern sorts out an input data collection \( A \) into an output data collection \( A' \) according to a function \( smaller(x, y) \) evaluating whether or not a collection item \( x \) is “smaller” than another collection item \( y \). Therefore, given a collection \( A = [a_1, \ldots, a_m] \) the pattern computes the collection \( A' = [a_{i1}, \ldots, a_{im}] \) such that in \( A' \) \( a_i \) comes before any \( a_j \) such that \( smaller(a_i, a_j) = true \).

Different kinds of parallelism may be exploited while computing the sorting pattern in parallel. Typically, a divide and conquer approach is followed where the collection to be sorted is split into smaller collections, each smaller collection is recursively sorted and eventually the sorted collection is build out of the sorted sub-collections.

**Parameters** The functional parameters of the sort pattern include (a reference to) the function judging whether a collection item is “smaller” than another one. The non functional parameters include the parallelism degree to be used to implement the parallel sort pattern [14]. In case the sort pattern is implemented by means of a proper divide and conquer pattern, all the functional and non functional parameters of the divide and conquer pattern are included in the functional and non functional sort pattern parameters.

As it could be expected, a sort pattern takes a collection of data as input argument and produce the sorted collection of data as output.

**Implementation** The sorting pattern is usually implemented by using a divide and conquer pattern such as the one described in Sec. 2.2.1 to implement a standard quick-sort parallel algorithm. Care is taken to consider suitable base cases, that is to avoid recurring too deep in the tree to avoid unnecessary synchronization and communication overheads while scheduling very small sort sub-tasks.

2.2.4 Pool evolution pattern

**Description** A “candidate selection” function \( s \) selects a subset of objects belonging to an unstructured object pool \( (P) \). The selected objects are processed by means of a “evolution” function \( e \). The evolution function may produce any number of new/modified objects out of the input one. The set of objects computed by the evolution function on the selected object are filtered through a “filter” function \( f \) and eventually inserted into the object pool. At any insertion/extraction into/from the object pool a “termination” function \( t \) is evaluated onto the object pool, to determine whether the evolution process has to be stopped or it has to be iterated.

A pool evolution pattern therefore computes \( P \) as result of the following algorithm:

1: while not(\( t(P) \)) do
2: \[ N = e(s(P)) \]
3: \[ P = P \cup f(N, P) \]
4: \textbf{end while}

**Parameters**  
The pool evolution pattern functional parameters include (references to) the candidate selection, evolution, filter and termination functions. Additional non functional parameters may be provided to set up the parallelism degree to be used in the computation of any one of these functions, notably the evolution and filter function.

Functional parameters may be used to implement less complex instances of the pool evolution pattern. As an example, an identity filter function simply inserts all the newly computed objects into the object pool, or a simple counting termination function iterates the pool evolution for a known number of times.

**Implementation**  
All the four phases of the pool evolution pattern are natural candidates for parallel execution. The selection function may be implemented in data parallel way, by partitioning the object pool and applying selection in parallel on the different partitions. The evolution function step is clearly an embarrassingly parallel one, as well as the filter process. The termination process may be implemented as a reduce pattern, but in some cases (e.g. when the “counting” termination function is used) it may be implemented as a plain sequential process.

### 2.2.5 Work-flow graph interpreter

**Description**  
A work-flow (WF) graph interpreter pattern models those parallel computations where the tasks to be computed as well as the relative ordering of the task computations are expressed through a work-flow graph \( G = (N, A) \). The set of nodes \( N \) in the graph are instructions, that is functions computing one or more outputs from one or more inputs. When the function code represent quite large portion of (sequential) code, the WF interpreter is also known as macro data-flow (MDF) interpreter. The set of arcs \( A \) in the graph defines the data dependencies among work-flow instructions. In particular, an arc leaving instruction \( WF_i \) to reach instruction \( WF_j \) represents an output produced by \( WF_i \) and consumed (as an input) by \( WF_j \). Only fireable instructions may be computed, that is instructions whose input data (input tokens) are already available/have already been produced by either other instructions or taken from the input stream(s).

Parallelism in the WF graph interpreter pattern comes from the concurrent computation of fireable instructions.

**Parameters**  
The WF graph interpreter pattern functional parameters include the WF graph to be executed and (a reference to) the code relative to the functions used in the different instructions in the graph. The non functional parameters include the parallelism degree of the parallel interpreter and, possibly, proper scheduling...
strategies to manage excess parallelism (cardinalities of the fireable instructions much larger than the amount of available parallel computing resources).

**Implementation** The WF graph interpreter may be logically implemented with a “matching unit” paired with a pool of instruction interpreters. The matching unit is in charge of dispatching fireable instructions to the parallel interpreter instances and of pairing the results computed by the interpreters with the proper target WF instructions. Each one of the instruction interpreters process a fireable instruction (a tuple with the function code and the function parameters (input tokens)) to produce a tuple of results, each directed to a specific input token position in another WF instruction.
2.3 Domain specific pattern set

2.3.1 Orbit pattern

**Description**  The orbit pattern comes from the *symbolic computing* domain and models the iterative construction of a set of items starting from an initial set \((S)\) of items using *generator functions* from a generator function set \((G = \{g_1, \ldots, g_k\})\). At each stage of the iterative process, the set of generators are applied to all the elements in the current set. The resulting set of new items is then added to original set taking care of avoiding duplicates.

Therefore the orbit pattern computes the transitive closure of a set according to a set of generator functions, and its computation is modelled by the following algorithm:

1: repeat
2:   for all \(x \in S\) do
3:     for all \(g \in G\) do
4:       \(New = g(x)\)
5:     for all \(n \in New\) do
6:       if not\((n \in S)\) then
7:         \(S = S \cup \{n\}\)
8:       end if
9:     end for
10:   end for
11: end for
12: until no more items added to \(S\)

**Parameters**  The functional parameters of the orbit pattern include (references to) the list of generators, while the non functional parameters include the parallelism degree. As argument, the pattern needs an initial item from which generating the output set.

**Implementation**  The orbit pattern is another example of an iterative pattern, since it assumes the generation of a number of items into a set, until no new additions occur. A possible implementation can be provided as a two stage process, the first stage producing the new items and the second stage filtering any possible duplicates. The first stage can be designed as a parallel stage of \(n\) replicas, each applied on a single item given as input to a (possibly parallel) pattern: such internal pattern evaluates all the possible values produced by the parallel application of the set of generator to the input value.

Another possible implementation is the one in which the filtering phase is partially implemented at the level of the first stage. In this case, a first filtering is operated immediately after the generation of new items and another stage is required in order to compare the partial filtered sets of items generated in parallel.
2.3.2 (Colored) Stencil Pattern

Description  Stencil computations occur in all those applications where items in a collection of objects dynamically change over a number of iterations or within a period of time, and each object change depends on the previous value of that object and on the value of the neighbour objects in the collection. The neighbour object set is called “stencil” and the shape and dimension of the stencil characterizes the stencil pattern. The neighbouring relationship (stencil function) can generally be described as a function $f_N : X \rightarrow \{X^*\}$ that maps each object into a set of other objects. The stencil function may be defined in the same way over all the elements of the input collection (static stencil) or may depend on the position of the items in the collection (dynamic stencil). Also, the stencil pattern may assume different stencils in different iterations. In this case we refer to “coloured” stencil, with colours denoting different iterations and, consequently different stencil patterns as well as different items in the input collection where the stencil is applied in a given colour iteration.

The stencil pattern models non embarrassingly data parallel computations and parallelism comes from parallel computation of disjoint sets of stencils within the same collection. Given an input collection $[a_1, \ldots, a_k]$, a stencil function $\text{Stencil}$, and a function $f$ computing the new items out of the neighbourhood, the stencil pattern computes, at each iteration, the collection

$$[f(\text{Stencil}(a_1)), \ldots, f(\text{Stencil}(a_k))]$$

Parameters  The stencil pattern functional parameters include the stencil shape (either position independent or position dependent) or a set of coloured stencil shapes (in case of coloured iterations) as well as (a reference to) the function computing the new item value out of the previous item value and of the values of the items in its stencil. The non functional parameters include the parallelism degree to be used in the computation of the stencil pattern.
As arguments, the a conveniently instantiated stencil pattern will take an input collection and produce an output collection as well.

Implementation  The implementation of the pattern can proceed in parallel for each item $X_i$ of the input collection. First the neighbour set of $X_i$ $\text{Stencil}(X_i)$ is computed, then the new value $X'_i$ is computed out of $X_i$ and of the values in $\text{Stencil}(X_i)$. Eventually, all the collection items ($X_i$) are updated to $X'_i$ in a barrier, and the process is repeated for the next iteration. Data parallelism is exploited while computing the new $X'$. 
2.3.3 Instances of work-flow graph interpreter pattern

2.3.3.1 Possibly iterated numerical library calls pattern

**Description**  This pattern models those applications where the code only contains loops and numerical library function calls. It is quite common in the domain of linear algebra in which algorithms are often built as a list of (possibly iterated) calls to numerical libraries such as BLAS and LAPACK functions. In particular, the pattern assumes a work-flow graph of the calls to the numerical library functions exist, either static or generated dynamically, that can be used to orchestrate parallel invocations of the numerical library functions with no dependencies.

**Parameters**  The functional parameters of the pattern include the list of the (references to the) numerical library functions to be called as well as the work-flow graph of the dependencies among these calls. The graph can be easily derived from any algorithm representing a numerical algorithm by considering the order in which the numerical library function calls are issued and the input/output parameter characterization of these function calls. A conveniently instantiated pattern of this kind will expect a stream of items (i.e. library calls) as input argument and the output is represented by the execution of the graph itself.

**Implementation**  The pattern is particularly suitable for being implemented exploiting a macro data-flow graph interpreter pattern such as the one described in Sec. 2.2.5, i.e. as a computing framework in which

- all the operations on data are considered as nodes $v_1, \ldots, v_i$ of a graph in which each edge $(v_1, v_2)$ between two nodes exists if $v_1$ and $v_2$ are functionally dependent, i.e. if $v_2$ needs some input values (called tokens) from $v_1$ at least, before it can be executed;

- all the nodes are eligible for a parallel evaluation (i.e. they are considered fireable and produce an output token by applying their representative library call over one (or more) input token), as soon as all their input tokens are available

- the first step of the pattern execution aims at building the data-flow graph, the second step interprets it by providing the initial input token and by executing in parallel the first set of fireable nodes;

2.3.4 Instances of the pool evolution pattern

2.3.4.1 Genetic Algorithm pattern

**Description**  Genetic algorithms offer a domain-specific abstraction for the already mentioned *Pool evolution pattern* when considering evolutionary computing
field. Generally speaking, it describes an iterative behavior in which, at each iteration step, a set of items (the individuals belonging to a population) evolves. The size of the population could change or could be statically defined. How the items change depends on the genetic operator that the pattern will apply (mutation and crossover, for instance), thus is a matter of application specificity.

**Parameter** From a functional perspective, a GA pattern takes a population as input and the number of iterations over which evolving the system. Depending on the specific algorithm, a set of functions (or even patterns) implementing the genetic operators could be required. As an example, in the Global Single Population Genetic pattern (see Section 2.3.4.2), an operator representing the selection phase is needed, at least while in the Evolutionary Multi-agent System the number of sub-population to be created is required in order to make the selection process more sophisticated. Moreover, non-functional parameters as the parallelism degree could also be required.

**Implementation** A number of implementation has been provided for this pattern in a quite specific manner, depending on the implementation of the population and, in particular, depending on which evolutionary strategies have to be represented. In the global single-population master slave GA [4] the pattern is instantiated as a master-slave skeleton accessing a shared data set representing the population. The evaluation of the fitness in done in parallel on each individuals, while the genetic operators can be implemented as sequential functions. In the single-population fine-grained GA skeleton, both the fitness and the genetic operators are executed in parallel, thus assuming some form of stencil among individuals of the same neighborhood. In the multi-deme coarse-grained GA skeleton, the population is splitted in sub-populations (called islands) that evolve independently until a certain threshold, after which some individuals migrate from his island to another in order to guarantee genetic change.

### 2.3.4.2 Global single population genetic pattern

**Description** The Global single population genetic pattern is a domain-specific instance of the Genetic Algorithm pattern (Sec. 2.3.4.1) where the evolution is a process involves the whole population in each generation. In fact, the population is seen as a single entity over which individuals evolve on the basis of a set of genetic operators. The population size tends to be statically defined, thus it does not change as the computation proceeds. The result of the global single population genetic pattern may be defined in terms of the algorithm computed by the pool evolution pattern algorithm (see Sec. 2.2.4)

1. `while not(t(P)) do`
2. \[ N = e(s(P)) \]
3. \[ P = P \cup f(N, P) \]
where $P$ represents the whole population, $s$ (the selection function) is the identity function (all the members of the current population are selected for possible mutations), $e$ is the function computing mutation(s) of an individual and $f$ is the function filtering the mutated individual with the best “fitness” scores.

**Parameters** The functional parameters of this pattern include (references to) the set of functions $e$ and $s$ representing the genetic mutation operators and fitness filtering functions to be applied at each iteration. The non functional parameters include the amount of iterations to be computed as well as the parallelism degrees to be used in the pattern computation. The input argument once the pattern has been instantiated is provided by an implementation of the population that could be represented as a stream of items or a collection of them. The pattern produce a stream of items (or a collection) representing the evolved population as output.

**Implementation** The pattern can be defined as an iterative process involving a sequence of stages in which the first stage evaluates the fitness of each individuals (which is typically a function provided by the programmer) in parallel. The selection function picks up a number of individual (typically, the ones exploiting the better fitness rate measured at the previous step) that will be modified by the application of some genetic operators (crossover and/or mutation, for example) in order to produce the next generation’s individuals which will replace the selected one. The process iterates, until the maximum number of generations has been produced.

### 2.3.4.3 Concurrent Memetization Operator

**Description** This pattern is also used in evolutionary computation in which iterative progress processing, such as growth or development in a population, is performed. With respect to other patterns in the family of genetic algorithm pattern, here the population is selected, during the iterative progress, using proper search operators in order to achieve the desired goal. The pattern involves continuous optimization and combinatorial optimization phases. It may be useful for implementing Lamarckian or Baldwinian memetic variation operators [5].

The procedure starts with a certain individual $i_{\text{init}}$ and a set of mutation operators ($M$) available. Then, according to the parameters, a series of mutations $m \in M$ and evaluations $f$ of new individuals is performed in parallel. The best found solution $i$ becomes a new starting point for another phase of the memetization. The best individual after the assumed number of phases is returned. The procedure can be defined by the following function:

1:  function $\text{STEP}(i_{\text{init}}, M, n)$
2:      $I = i_{\text{init}}$
3:      $V = \langle i_{\text{init}}, f(i_{\text{init}}) \rangle$
for i=1 to n do
  for all i ∈ I do
    for all m ∈ M do
      new = m(i)
      I = I ∪ {new}
      V = V ∪ {⟨new, f(new)⟩}
    end for
  end for
  i = reduce(best_of, V)
return i

Parameters  The functional parameters of the pattern include the set of mutation functions M and the number of iterations to be implemented. The non functional parameters include the parallelism degree to be used in the computation of the pattern. Once the pattern has been instantiated, it takes an item (i.e. a single individual) as input and returns an item (i.e. the better evolved individual) as output.

Implementation  The pattern represents an iterative procedure starting from an individual on which, in parallel and for each iteration, a number of actual mutations and evaluations of the fitness function are performed. A reduce pattern allows then the generation of the best individual and the procedure starts again with new mutations/evaluations. The process is finished after reaching the maximum number of phases, defined as one of the parameters. The process returns the best individual found up to date.

The pattern may be clearly be implemented exploiting a pool evolution pattern such as the one discussed in Sec. 2.2.4 using identity as the selection function, the mutation functions as the worker functions, a null filter function and a counter termination function. The pool pattern should be followed by a reduce pattern computing the “best” mutated individual in the individual result collection.
Chapter 3

Other patterns

This part of the deliverable hosts the description of a number of patterns that have been proven useful in different applicative domains but that have not been (planned to be) used in the project use cases and applications. As stated in the initial part of this document, these patterns will not be implemented within ParaPhrase, unless at a given point they do not become relevant for the efficient implementation of some project use case.

3.1 MapReduce

Description  The MapReduce pattern models those applications where a collection of input data is processed in two steps: 1) a map step computes a \( \langle \text{key}, \text{value} \rangle \) pair for each item in the input collection by applying a function \( f \), and 2) a reduce step “sums up” all the value items with a given key (in parallel for all keys) using an associative and commutative operator \( \oplus \).

The result computed by the MapReduce pattern (MR) may be described as follows. Given an input collection \( (x_1, \ldots, x_n) \), \( x_i \in X \), a function \( f : X \rightarrow \langle \text{Key}, \text{Y} \rangle \), a binary and associative function \( \oplus : Y \rightarrow Y \rightarrow Y \) and assuming that the function \( \text{keys}_o f : \langle \text{Key}, \text{Y} \rangle \text{list} \rightarrow \text{Key list} \) returns the list of keys appearing in the first list of pairs and that the function \( K : \text{Key} \rightarrow \langle \text{Key}, Y \rangle \text{ list} \rightarrow \text{Y list} \) returns the list of values of the items in the \( \langle \text{Key}, Y \rangle \) list with a given key, then

\[
\text{MR}(f, \oplus)(x_1, \ldots, x_n) = \{ \Sigma_{\oplus}(K(k)) \mid k \in \text{keys}_o f(f(x_1), \ldots, f(x_n)) \}
\]

The parallel computation of \( \text{MR}(f, \oplus)(x_1, \ldots, x_n) \) exploits data parallelism in the computation of \( f \) over all the elements of the input collection—the input collection is usually already partitioned across the set of available processing elements—and in the set of reduces that are cooperatively executed by the same processing elements immediately after having computed the \( f \) applications in parallel. It is worth pointing out that MR is different from the simple composition of map and reduce patterns described previously.
Parameters  The functional parameters of the MapReduce pattern include (the references to) the functions \( f \) and \( \oplus \). The non functional parameters include the parallelism degrees used in the map and reduce phase of the pattern. The input argument of such an instantiated pattern is represented by a collection of item and the output is given by a single item providing the reduction.

Implementation The input collection is typically already partitioned among all the available processing elements. The MR pattern is typically implemented using a set of PEs which compute the \( f \) function independently over the local partition of the input collection. The \( \oplus \) function is then applied on the local results of the \( f \) function and the partial results produced are re-distributed among the available processing elements in order to have each processing element computing the final reduce value relative to given key (or key set).

A well-know implementation of this pattern is the Google MapReduce [8].

3.2 Generalized zip

Description The generalized zip pattern combines two or more “isomorphic” (structurally identical) and same size data structures by merging corresponding elements using a function \( f \).

More formally, given \( m \) input collections each one with \( n \) elements

\[
X_1 = [x_1^1, \ldots, x_1^n], \ldots, X_m = [x_m^1, \ldots, x_m^n]
\]

and such that the type of elements in collection \( X_i \) is \( \alpha_i \) and a function \( f: \alpha_1 \times \ldots \times \alpha_m \rightarrow \beta \), the generalized zip pattern computes in parallel the output collection

\[
Y = [y_1, \ldots, y_n]
\]

with elements of type \( \beta \) where

\[
y_k = f(x_1^k, \ldots, x_m^k)
\]

As in the map pattern, parallelism in this pattern comes from the execution of the different \( f(x_1^k, \ldots, x_m^k) \).

Parameters  The functional parameters of the generalized zip pattern include the (reference to the) function \( f \) while the non functional parameters include the parallelism degree used to evaluate in parallel all the result items. This pattern takes two data collection as input argument and produce a single data collection as output.

Implementation The generalized zip pattern is typically implemented using a set of PEs which compute the \( f \) function independently over a contiguous partition of the input collections. The output produced by each PEs is already the final result without needing any further step. The Muesli [16] zip skeleton is an example of the merge pattern applied to an input collection of 2 data structures.
3.3 Scan

Description The scan pattern (also known as all-prefix-sum), models those computations where an input collection of \( n \) elements is used to produce an output collection of \( n \) elements with each element being the “sum” of the previous elements of the input collection, i.e. all elements with a lower index value. The inclusive scan adds the value of the element in the “sum”, while the exclusive scan does not. The “sum” here is computed with a binary, associative and commutative operator. Given a binary associative operator \( \oplus \) and a collection \( X \) of \( n \) elements \([x_1, \ldots, x_n]\) the result of the \( \text{scan}(X, \oplus) \) is

\[
[x_1, (x_1 \oplus x_2), \ldots, (x_1 \oplus x_2 \oplus \ldots \oplus x_n)]
\]

As an example, given the initial collection of \((1, 2, 3, 4, 5)\) and considering as \( \oplus \) function the addition operator, the inclusive prefix scan of the initial set is \((1, 3, 6, 10, 15)\) whereas the exclusive one is \((0, 1, 3, 6, 10)\).

Parameters The functional parameters of the scan pattern include the (reference to the) function \( \oplus \) which has to be binary associative. Non functional parameters include the parallelism degree used to compute the scan. A conveniently instantiated scan pattern will take a collection as input argument and will produce a singleton as output.

Implementation The reference implementation of the scan pattern is presented in [3]. Given \( p \) processing elements (PEs) and an input collection \( n \) of \( n \) elements, the algorithm first divides the sequence in \( p \) parts and assigns each part to a distinct PE which then sums up all the elements in the segment (reduce phase). The result computed in the reduce phase “is sent” to the neighbour PE, i.e. \( k \to k + 1 \) \( k \in (1, p) \) which adds the value “received” to the value computed in the reduce phase and sends it to the neighbour (up-sweep phase). After having sent the value to the neighbour, each PE adds the value received to the first element of the local sequence (all but the first PE) and then starts adding each number to its successor in the segment assigned to it (down-sweep phase).

3.4 Fork/Join

Description The fork/join pattern models those computations where a control flow sparks a number of new control flows and awaits for the termination of all of them before continuing its execution. Each one of the new control flows proceeds independently and eventually terminates. Different variants of the pattern exists, depending on the ways supported to move data to and from the new control threads. Usually, either the possibility to share data or to exchange data through explicit send/receive communication primitives is considered.
Parallelism in the fork/join pattern comes from the parallel execution of all the new control flows.

**Parameters** The functional parameters of the fork/join pattern include (references to) the patterns that have to be forked. Non functional parameters may include the parallelism degree. By default the parallelism degree is equal the number of forked control flows. Through the non functional parameter it may be reduced to a smaller value and in this case multiprogramming may be used to execute the different control flows on the available resources.

**Implementation** The fork/join pattern is a kind of low level parallel pattern [14]. The implementation is quite trivial, both in case threads are used to spark the new control flows and in case processes are used instead. All interactions among the new control flows and between the new control flows and the control flow that actually initiated the fork/joint pattern are completely in charge of the programmer, and awaiting the termination of a set of either threads or processes is usually a kind of library call primitively provided by any thread/process framework.

### 3.5 Iterated Map with Dependency Tracking

**Description** The pattern models those computations where a dependency graph is updated taking into account the encoded dependencies. Given a graph $G = (N, A)$, a function $f : N^* \rightarrow N$, a function $g : A \rightarrow A$ and a subset of the nodes $I \subseteq N$ the iterated map with dependency tracking computes a new graph $G' = (N', A')$ iteratively as follows:

1. $I' = I$
2. repeat
3. set $G = G'$
4. for all $n_i \in I'$: do
5. $S = \{ n \in N \mid (n_i, n) \in A \}$
6. change $n_i$ in $N'$ to $f(n_i, S)$
7. $I' = I' \cup \{ f(n_i, S) \}$
8. $U = \{ (n_i, x) \in A \}$
9. $A' = (A' - U) \cup g(U)$
10. end for
11. until $G' \equiv G$

At each iteration, the set of candidate nodes $I'$ is considered. Each node is updated according to the function $f$ taking as inputs the node and all the nodes connected to that node. The arcs (dependencies) insisting on the node are updated according to the function $g$. In principle, arcs insisting on the node may be cancelled or added by $g$. Eventually the process is iterated up to the point the whole graph is updated and no more arcs and nodes are modified.
Parallelism in the pattern comes from the concurrent evaluation of the changes induced by the items in the set $I'$.

**Parameters** The functional parameters of the pattern include the (references to the) functions $f$ and $g$. The non-functional parameters include the parallelism degree.

**Implementation** The pattern is implemented by iteratively evaluating in parallel the effect of the changes induced by the nodes in the $I'$ set.

### 3.6 Branch&Bound

**Description** The Branch&Bound (B&B) pattern is used to find optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. The algorithm modeled by the pattern consists in a systematic enumeration of all possible solutions for a given problem, where large subsets of non-optimal solutions are discarded (pruned) by using upper and lower bound estimates of the quantity being optimized.

The associative search over a set of items that matches some input criteria, can be considered an example of the B&B algorithm in the case that only one result that satisfies the criteria is needed. The parallel search can be performed splitting the input set in partition and searching the item in each subset in parallel. Once an item matching the search criteria is found in any one of the parallel subset searches, the searches in the other subsets are cancelled. If there are multiple possible matches, this pattern is non-deterministic because which match is returned as final result depends on the timing of the searches over each subset.

The B&B algorithm can be easily described as a recursive algorithm: a given problem is recursively divided into a number of subproblems and lower bounds for the optimal solution of each subproblem are computed. If a solution of a subproblem is found, this solution is also a solution of the overall problem. All other subproblems can be discarded, if the corresponding lower bounds are greater than the value of the current solution. Subproblems with smaller lower bounds still have to be considered recursively.

Two basic approach for the exploration of the solution area can be distinguished:

1. **BestFirst**: This approach assumes that the objective function has to be minimized. At every step the solution with the minimal lower bound is improved. A disadvantage of this approach is the high memory usage because all solutions not discarded have to be maintained in memory.

2. **DepthFirst**: This approach improves every solution till a better solution is found or the lower bound of the current solution is higher than the upper
bound of the current best solution. This method uses less memory but requires larger computation times.

A B&B problem can be solved by applying a small set of basic rules. While the signature of these rules is always the same, the concrete formulation of the rules is problem dependent. Starting from a given initial problem, subproblems with pairwise disjoint state spaces are generated using an appropriate branching rule. A generated subproblem can be estimated applying a bounding rule. Then, using a selection rule, the subproblem to be branched from the next is chosen from a work pool storing the subproblems generated. Subproblems with non-optimal or inadmissible solutions can be eliminated during the computation using an elimination rule.

**Parameters** The functional parameters of the B&B pattern include a reference to the branching function which makes a partition of the current solution set, a reference to the bound function which estimates lower and/or upper bound of the generated possible solutions (subproblems), a reference to the selection function which selects the next subproblem to be branched and a reference to the elimination function which discard non-optimal solutions found during the computation. The non functional parameters of the pattern include the parallelism degree. One instantiated, the pattern takes a representation of the problem as input argument (a collection of data, for instance) and returns the better “solution” as output.

**Implementation** Parallel implementations of the B&B algorithm can be classified depending on the organization of the work pool used to store the subproblems generated during the computation. A centralised, distributed, and hybrid organization of the work-pool can be used. The Muesli skeleton library [16] offers a distributed implementation of the B&B pattern.
Bibliography


