Project no. 288570

PARAPHRASE

Strategic Research Partnership (STREP)  
PARALLEL PATTERNS FOR ADAPTIVE HETEROGENEOUS MULTICORE SYSTEMS

Application-Specific Patterns Report  
D2.3

Due date of deliverable: 30th November 2012

Start date of project: October 1st, 2011

Type: Deliverable  
WP number: WP2  
Task number: T2.1

Responsible institution: UNIPI  
Editor and editor’s address: Sonia Campa, UNIPI

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
<td></td>
</tr>
<tr>
<td>PP Restricted to other programme participants (including the Commission Services)</td>
<td>√</td>
</tr>
<tr>
<td>RE Restricted to a group specified by the consortium (including the Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium (including the Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary

This is the third deliverable of the Work Package 2 Parallel patterns. It provides the description of an initial set of domain specific parallel patterns. These patterns have been identified by analyzing the requirements provided by the different “application” communities contributing to the project. In particular, three different application domains have been considered, namely symbolic computation, numerical computation and soft computing domains.

For each domain, an initial set of domain specific patterns has been identified. Within each set of domain specific patterns a “representative” pattern has been identified and its characterizing properties have been described using the methodology of the software engineering parallel pattern community. Last but not least, for the identified representative patterns we have outlined possible implementation strategies using the core, “RISC” patterns introduced in D2.1 (the “Initial generic patterns report” [18]) and whose implementation is discussed in D2.2 (“Homogeneous implementation of initial generic patterns” [17]), which constitute the “basic building blocks” of parallel applications according to the general ParaPhrase project design. These implementation strategies are aimed at establishing a general domain specific pattern implementation methodology suitable for supporting the implementation of any of the domain specific patterns considered within the project.

It is worth pointing out that, as the project evolves, we expect to extend this first set of domain specific patterns with new patterns suggested from the project beneficiaries contributing to the use case work package (WP6).

As far as this deliverable is concerned, we will use C++/FastFlow code snippets to illustrate the implementation strategies of domain specific patterns. As the code refers to the same skeletons implemented in Erlang, the overall methodology naturally extends to the Erlang ParaPhrase framework.

The placement of this deliverable within WP2 and, more generally, within the project, is illustrated in Fig. 1, on the following page.
Figure 1: Positionining of Deliverable D2.3
# Contents

- Executive Summary ...................................................... 1  
- **Introduction** ........................................................... 4  
  - **Application specific patterns** ....................................... 8  
    - 2.1 The Global Single Population Genetic pattern ................. 8  
    - 2.2 Orbit pattern ...................................................... 11  
    - 2.3 Possibly Iterated Numerical Library Calls pattern .......... 13  
    - 2.4 (Coloured) Stencil pattern ...................................... 16  
- **Conclusions** ............................................................ 19
Chapter 1

Introduction

The methodology proposed by ParaPhrase focuses on the introduction and exploitation of parallelism through the use of compositions of parallel design patterns, eventually compiled to suitable compositions of algorithmic skeletons. In Deliverable D2.1 [17] a programming methodology was proposed providing three levels of abstraction for the definition of a parallel application:

- at the higher level, an application is defined in terms of a composition of parallel patterns. Each pattern represents a particular model of parallel exploitation that is either data or stream-parallel. A pattern is defined in terms of a name and a problem it can solve, i.e. through a natural language description of the problem tackled and of the associated solution technique(s).

- at a lower level (the high level implementation level) each pattern is associated with an abstract skeleton or, when more complex patterns are considered, with a composition of abstract skeletons. Abstract skeletons are constructs provided to the programmer to implement parallel patterns. Each abstract skeleton is described by a name, a functional behavior, a parallel activity graph (i.e. a description in terms of computational nodes) and a signature.

- at the bottom level, (the low level implementation level) the actual implementation of the abstract skeletons is exposed relatively to the target architecture. For the purposes of the ParaPhrase project we will provide implementations of the skeleton set in C++ and in Erlang.

While defining the parallel patterns/skeletons, in deliverable D.2.1 [18] we proposed to classify them in terms of their computational complexity as:

- core skeletons/patterns, representing the basic building block of a parallel computation, that is a minimal set of quite simple as well as basic parallel exploitation skeletons/patterns supporting composition and such that they can be used, alone or in composition, to describe a wide range of complex patterns. In this set we included: pipe, farm, seq, map, and reduce.
• **high-level skeletons/patterns**, modeling very general parallel patterns appearing in domain specific contexts and closer to the application programmer’s viewpoint and programming habit with respect to the core skeletons/patterns.

Deliverable 2.1 provided an overview of the scenario and presented the definition of the initial set of high level patterns. Subsequently, Deliverable 2.2 discussed the homogeneous implementation of the initial generic patterns with respect to the core skeletons while this Deliverable will focus on the definition of high-level skeletons/patterns, in particular of those specific to a given application domain.

We therefore considered three different application domains, selected on the basis of the peculiar experiences proper of the ParaPhrase beneficiaries. The three domains we considered are:

1. **soft computing**: applications belonging to this field approach the problem of exploring wide and very complex solution spaces by emulating the decision processes of biological systems. Evolutionary computing, with its set of genetic algorithms, is a branch of this field in which the optimization techniques are inspired by biologic evolution theory: in fact, the strategies used to explore the solution space are based on the concepts of selection, mutation, crossover and fitness, evaluated for a population (representing the input data set) over the evolution of several generations.

2. **linear algebra**: in this field of the mathematics, linear phenomena are described in terms of vector spaces whose properties are investigated by systems of linear equations. This field embraces also the study of nonlinear mathematical models that can be approximated by linear ones and its techniques find application in wide range of scientific fields including physics, computer science and natural sciences.

3. **symbolic computation**: this field is related to the processing of symbolic—that is, non numeric—information, which is typical of the kind of information processed by compilers/interpreters, for example. The symbolic information is usually processed using dynamic data structures, quite different from the data structures needed to host data in numerical applications.

Within these domains, we looked for well-know and widely-used (parallel) patterns with a view to implementing these patterns according to the ParaPhrase methodology to provide effective and productive programming frameworks to programmers working in these application domains.

For each of the domains, we identified several domain specific patterns, which are listed in Table 1.1. These lists are subject to extension/changes during the progress of the ParaPhrase project as a consequence of further investigation by the beneficiaries working on the use case applications (WP6).

From each list, that is from each set of patterns specific to the same domain, we selected a “representative” pattern. For this representative pattern, we present
in Chap. 2 a description as typically provided in the software engineering community [11]. Within the description, we outline a possible implementation of the representative pattern in terms of core patterns (and therefore core skeleton compositions).

The main aim behind the selection of the representative patterns and to the outline of a possible implementation in terms of core patterns is to design and illustrate an effective methodology suitable for guiding the system programmer in the process of implementing new domain specific patterns within the ParaPhrase programming framework. This has been preferred to an exhaustive description of the full set of patterns listed for the different application domains to better point out the layered approach followed in ParaPhrase to support parallel application development:

- high level, domain specific patterns are directly provided to the application programmer in order to improve his/her productivity
- high level patterns are compiled in terms of (compositions of) abstract skeletons
- the skeletons are implemented (and optimized) for the different “target platforms” namely C++/FastFlow and Erlang in ParaPhrase.

It is worth pointing out that different application/domain specific patterns may model the same pattern in different contexts. In other words, conceptual overlaps exist among domains so that patterns defined inside a specific domain could be useful in other domains. This feature underlines once more the abstraction power of the ParaPhrase approach: patterns represent how the parallel computation is structured and organized, interpreting elaborative behaviours that can be generalized with respect to specific context or domains by compiling the patterns to compositions of general-purpose, domain-neutral skeletons. For example, the Iterated Map with Dependency Tracking pattern works on a graph by progressively updating a node and its connected nodes in a pattern evoking the transitive closure needed in many applications of the Soft computing domain. On the other hand, the transitive closure pattern can be seen as a generalization of the Orbit pattern coming from the same field and discussed in Chap. 2.
<table>
<thead>
<tr>
<th>Domain</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soft computing</td>
<td>dynamic programming patterns [13], multi-objective optimization evolutionary patterns [6], global single population genetic patterns [4], single population fine-grained genetic patterns [12], multi-deme genetic patterns [1], Iterated Map with Dependency Tracking pattern, Computational Object Pool Pattern</td>
</tr>
<tr>
<td>Linear Algebra</td>
<td>Possibly iterated numerical library calls [9], Ghost Cell Pattern [10], Basic stencil pattern, Cache optimized stencil, Coloured stencil</td>
</tr>
<tr>
<td>Symbolic computation</td>
<td>Orbit [14], transitive closure pattern [14], duplicate elimination pattern [7], completion algorithm pattern [7], chain reduction pattern, critical-pair-completion pattern, multiple-homomorphic-images pattern [15]</td>
</tr>
</tbody>
</table>

Table 1.1: Reference table of domain specific patterns
Chapter 2

Application specific patterns

For each of the domains listed in Tab. 1.1 we have selected one or more representative pattern and here we provide a full description of each of them. For the soft computing domain, in Sec. 2.1 we introduce and describe the Global Single Population Genetic pattern already introduced in D2.1. For the symbolic computation domain, in Sec. 2.2 we describe the Orbit pattern modelling the exploration of a solution space starting from a set of initial values and using a number of “generating functions” to generate (and therefore explore) the complete space. Finally, for the linear algebra domain, in Sections 2.3 and 2.4 we will discuss two domain specific patterns:

- In Sec. 2.3 we will discuss how the Cholesky block decomposition algorithm can be formalised in a parallel pattern named Possibly Iterated Numerical Library Calls pattern and how it can model a way to solve a system of linear equations. It is worth pointing out that this is a very wide domain and that the implementation of the block Cholesky algorithm requires the use of advanced, external numerical libraries in order to ensure performance.

- In Sec. 2.4 we will discuss the (Coloured) Stencil Pattern which is a very useful domain specific pattern relatively to the applications developed within WP6. that provides a well established neighbourhood computation example. This pattern represents a generalization of classical data parallel “stencil” computations factorizing the “neighbourhood” elements through a proper functional parameter rather than computing the set of neighbours just relying on “index proximity”.

2.1 The Global Single Population Genetic pattern

Problem The global single population genetic pattern manages the evolution of a set of items up to a termination condition. The convergence is evaluated along a certain computational period that can be modeled on the basis of different parameters (for example, the number of iterations as well as the consumption of the
Context The global single population genetic pattern belongs to the soft computing application domain and, specifically, to the evolutionary computation field. Evolutionary algorithms implement optimization techniques by taking inspiration from biological evolution such as selection and mutation process in order to investigate a solution space. The basic idea is that individuals of a population (i.e. the solution space) improve their fitness generation after generation while converging to the optimal. The improvements depend upon the individuals selected for reproduction/mating and upon guaranteeing genetic diversity which is introduced by slightly modifying selected individuals inside the population, i.e. emulating the mutation process. An evolutionary algorithm stops when a satisfying level of convergence is reached and/or the population is extinct or a number of generations have been produced. The criterion is typically defined by the programmer.

Forces This optimization pattern is very general and represents an optimization strategy often adopted in other application domains (numerical analysis, artificial intelligence, neural networks, machine learning, to name but a few) [6, 13]. Moreover, the pattern can be very flexible because its functional behavior can radically change depending on how the population is defined/implemented and which criteria are introduced through the genetic operators involved.

Solution The definition of the pattern depends on (at least) four parameters:

• a pool representing a population
• a selection function of individuals on which the evaluation (the convergence function) has to be applied
• a function to be applied to the selected individuals; each individual could
  – be consumed (die)
  – return to the population (in modified or unmodified state, i.e. mutated)
  – be replicated in the form of (several) descendants (reproduction)
• a termination function taking a pool and returning a boolean value (true if and only if the algorithm converges).

The selection can be based on the random selection of individuals in the population or by assigning them specific values (“a fitness”) and picking up the better ones. Thus, the behavior of the selection has to be customizable. On the other hand, the termination function could wait for the extinction of the population or it could be based on the (possibly parallel) evaluation of the fitness/quality of the whole population or of a part of the individuals.

Functionally speaking, the behavior of the pattern could be formalized as follows:
while termination() do {
    selection(pool);
    par_eval(pool);
}

where termination() and selection() are sequential functions and par_eval is a farm in which each selected individual is evaluated by a worker. The definition of the population (pool) is requested from the user. Note that termination() could also be modeled as a farm and selection() could be a farm “labeling” some individuals; the whole pattern is depicted in Fig.2.1. The activity graph defining the skeleton associated with the pattern can be described as a three stages pipeline with feedback iterating over successive generations. Since the selection function is responsible for deciding which individuals of the population will be evaluated for a given generation, it can easily provide the behaviour of the farm by instancing the par_eval behaviour function. Structurally speaking, the skeleton appears as a two-stage pipeline in which:

- the first stage evaluates the termination condition
- the second stage is a farm in which the emitter selects a subpopulation to be evaluated by the workers; each worker applies an evaluation function to its assigned individual (or partition of them); the collector collects the new individuals (if generated) and provides a feedback to the termination node.

The skeleton is depicted in Fig. 2.2 and can be formalized as follows:

\[ T = Seq(termination()) \]
\[ W = eval() \]
\[ E = selection() \]
\[ ObjPool ::= pipe(T, farm(E, W)) \]
where $T$ is a skeleton (a Seq in this instance) implementing the termination condition; the second stage of the pipe is a farm for which $E$ defines the selection function that will be executed by the emitter and $W$ defines the (possibly sequential) function executed by the workers.

**Examples** A possible API for this pattern in C++ could be used as follows:

```cpp
for(int i=0; i<PARDEGREE; ++i)
    w.push_back(new Evaluator(i));

ff_objpool pattern(new Termination(), new Selector(), w);
```

where `ff_objpool` is the class defining the skeleton whose parameters are the termination, the selection and the evaluation functions that will be composed in order to build the two-stage pipeline.

We have to point out that the actual behavior of each node depends on the implementation of the population. How the population is implemented (for example, a collection of unordered items or a priority queue of values) could influence the selection policies and, as a consequence, how the workers access the shared dataset. The implementation of the population, which is left to the programmer, is non-trivial and an important aspect for the actual overall performance.

## 2.2 Orbit pattern

**Problem** The exploration of a solution space is required starting from an initial value and a number of generating functions.
Context  The Orbit pattern is quite common in the field of symbolic computation [7, 14–16] and is based on the generation of new elements belonging to a solution space; generation proceeds until no more elements can be generated. The input of the pattern is represented by a list of generators and a starting element in the domain of the application. Iteratively, new elements in the space are generated by applying the set of generators to all the elements already produced. The expected output is a list of generated elements belonging to the solution space. Its behaviour can be described as follows:

fun Orbit (input: list of generators, elem pt; output: list of elem){
    q = [pt];
    output = [pt];
    forall x in q do
        forall g in list of generators do
            y = apply(g, x)
            if not y in output
                q = q::[y];
                output = output::[y];
    }

Forces  The behaviour of the pattern is embarrassingly parallel in the elements of the local data structure q and in the number of generations, and so it seems particularly applicable in a parallel exploitation. However, care should be taken in the update of both the output set output and the local data q, in order to avoid
the insertion of replicas of the same element.

**Solution** The *Orbit* pattern can be easily described as a *farm* applied at two levels of parallelism. For each element \( q_i \) of the solution space the set of generators \( \{g_1, \ldots, g_k\} \) could be applied in parallel in order to produce new elements. This exploitation can be represented by a *farm* broadcasting \( q_i \) to \( k \) workers \( w_1, \ldots, w_k \) (one for each generator). Each worker \( w_j \) applies \( g_j \) to \( q_i \). Besides, all the elements in the solution space could be evaluated in parallel by an instance of such *farm*.

Thus, globally speaking the pattern is described by a two-layered farm as depicted in Fig.2.3 and formalized as follows:

\[
W ::= \text{Seq}(f) \\
F ::= \text{farm}(W) \\
Orbit ::= \text{farm}(F)
\]

where \( f \) represents a sequential code function applying a generator to an item and updating the accumulated sets.

The skeleton terminates when no more new elements can be generated.

**Examples** The definition of a skeleton representing the Orbit pattern requires the specification of:

- a set of \( k \) functions that take a value and return a new generated one on the basis of their assigned generator \( g_j \)
- a function implementing a filter criterion, i.e. a function that checks if a value to be inserted into the data set is already available

### 2.3 Possibly Iterated Numerical Library Calls pattern

**Problem** The pattern captures the efficient execution of a numerical algorithm expressed as (possibly iterated) sequences of (perhaps nested) loops and calls to numerical libraries.

**Context** In the context of linear algebra, a great number of algorithms are structured as a sequence of calls to computationally intensive numerical libraries. Such calls can be part of the body of a loop and loops can be nested. Well known problems such as *LU* or *QR* factorization and *Cholesky decomposition* exploit this kind of computational pattern, calling standard BLAS and LAPACK functions.

**Forces** The pattern of computation is very general and applicable to those cases in which the algorithm is defined by a *well-formed code*, i.e. code containing only loops and functional calls.
**Solution**  The pattern required to solve the problem is a data-flow graph interpreter. Data-flow is the well-known computing model [5, 8] where the execution of the program depends on data dependencies among functions. Thus, a program is represented by a graph in which nodes represent instructions (i.e. functions to be evaluated) and arcs among nodes represents data dependencies among them. The execution of a data-flow program starts with the assignment of the input data (“token” in data flow jargon) to the input arcs of the first instruction in the graph and proceeds with a loop. The loop body looks for “fireable” instructions in the data-flow graph—instructions with all the input data available (all input arcs with a token present)—and executes these instructions, possibly in parallel. The results of the fireable instructions executed are then directed to the destination data-flow instructions identified by the output arcs of the instruction, and the loop is restarted. The program terminates when there are no more fireable instructions or data tokens on arcs directed to other instructions.

In terms of skeletons, the interpreter for a data-flow graph can be seen as a farm with loopback [3]. The workers execute “fireable” executions and receive tasks from the emitter as they are available and send them back to the emitter that checks which new instructions can be executed. The program terminates when no more tasks need to be dispatched among the workers.

**Examples**  The block Cholesky factorization algorithm provides an example belonging to the linear algebra domain in which the proposed pattern can be successfully exploited [2]. The pseudo-code for the left-looking variant of the block Cholesky factorization algorithm of a matrix $A$ of complex numbers can be written as follow using BLAS Level 3 routines (i.e. CGEMM, CHERK, CTRSM) and Lapack auxiliary routine (i.e. CPOTF2):

```plaintext
for k=0.. N-1
    for n = 0.. k-1
        A[k][k] := CHERK(A[k][n], A[k][k])
        A[k][k] := CPOTF2(A[k][k])
    for m = k+1.. N-1
        for n=0..k-1
            A[m][k] := CGEMM(A[k][n], A[m][n], A[m][k])
            A[m][k] := CTRSM(A[k][k], A[m][k]);
```

The corresponding data-flow graph of the above algorithm can be derived as sketched in Fig. 2.4 for a matrix of $4 \times 4$ blocks. In this case the macro data flow instruction is represented by the following tuple:

$$\langle Id, Op, (i, j, k) \rangle$$

where:
Figure 2.4: The data-flow graph (left) and the instruction table (right) for the $4 \times 4$ block Cholesky factorization algorithm.

- $Id$ is the instruction id, that is a unique reference.

- $Op$ is one of the possible (macro)-instructions among CPOTF2, CTRSM, CHERK, CGEMM.

- $(i, j, k)$ are the values of the corresponding indexes that have to be used for computing the operation $Op$.

All data flow instructions can be extracted for a given factorization problem by simply visiting the graph top-down. In Fig 2.4 (left-hand side) is shown the instruction table for the case $4 \times 4$ blocks. The instruction table contains also, for each instruction, the number of data dependencies (the Dep. column) and the list of the next fireable instructions.

The skeleton implementing the pattern, simply schedules the instructions contained in the data flow instructions table according to the data dependency values. If the value becomes or is equal to zero then all the instructions in the instructions list can be scheduled to be executed in parallel. As soon as one instruction has completed its execution, the dependency value in the table is atomically decremented by 1.
2.4 (Coloured) Stencil pattern

**Problem** Given a set of objects \( \{X_1, \ldots, X_n\} \), a stencil, i.e. a function \( f_N : X \rightarrow \{X^*\} \) mapping each object into a set of neighbouring objects, and an interaction \( f_F : X \times X \rightarrow \mathcal{R} \), iteratively compute for each object the effect of \( f_F \) with all its neighbour objects, that is \( \forall X_i \) compute

\[
\sum_{j \in f_N(X_i)} f_F(X_i, X_j)
\]

**Context** Stencil computations are common in a number of different areas. They are used to solve differential equation systems, as well as to model different real systems, such as bodies subject to gravitational force, oil percolation within porous material, fluid flow on wing surfaces, etc. Stencil computations are important in image processing as well, as the convolution operation—which is basically a stencil computation—is fundamental in a number of different image processing algorithms.

In most cases, the effect of the interactions of the neighbouring objects consists in a change in the object properties (e.g. coordinates in space, shape, potential, etc.) and the computation has to be performed iteratively until a given convergence criterion has been reached.

Different kinds of stencil patterns are characterized by static or dynamic stencils. In static stencils, the function \( f_N(X_i) \) does not depend on the positioning of \( X_i \) in the object set. In dynamic stencils, the \( f_N(X_i) \) also depends on \( i \).

**Forces** Stencil operations may exploit regularity in the neighbourhood computation and therefore they may be efficiently implemented with techniques similar to the ones used when implementing more classical map/forall/embarrassingly parallel data parallel patterns.

In particular, different kinds of optimization may be performed if the stencil is static and the \( X \) object data may be stored in shared memory, according to the principle of the *owner-computes-rule*. In the case of distributed memory architectures, possible optimizations include communication factorization.

Stencil applications in general present huge opportunities for parallelism, and actually some of the benchmarks used to evaluate performance on the Top500 machines use stencil parallel patterns.

**Solution** A stencil computation may be easily expressed in terms of map and reduce patterns. Let us assume our stencil pattern is expressed as

\[
\text{stencil}(f_N, f_F, \oplus, X)
\]

then it may be implemented as three steps:

1. First, with a map pattern, for any \( X_i \) we generate the set of neighbours \( F_N(X_i) = \{X_{i1}, \ldots, X_{ik_i}\} \)
2. Second, with a map pattern, for each one of the pairs \( \langle X_i, \{X_{i1}, \ldots, X_{ik_i}\} \rangle \) we compute the set \( \{f_F(X_i, X_{i1}), \ldots, f_F(X_i, X_{ik_i})\} \).

3. Third, with a map(reduce) we compute for each \( X_i \) the effect of the interaction force \( f_F \) as \( f_F(X_i, X_{i1}) \oplus \ldots \oplus f_F(X_i, X_{ik_i}) \), where \( \oplus \) is the function "summing up" the effect of the interactions of the different neighbours with the object \( X_i \).

The efficiency of this implementation may be increased in different ways. We mention two distinct cases:

- When targeting shared memory architectures, the \( X_i \) objects are stored in shared memory. Each parallel worker is given a subrange of the objects (i.e. a partition of the overall object pointer set). Each worker then computes in parallel with the other workers the effect of interaction forces relative to the object partition it has been assigned. The object set is tentatively partitioned in such a way that all the items assigned to a worker are stored in the fastest memory bank.

- When targeting distributed memory architectures, the set of objects is partitioned among the available workers. Before starting an iteration of the stencil computation, the objects not stored locally in the worker memory but necessary for the computation of the new state of the local objects are gathered w.r.t. the core where the worker is being run.
from their “home” worker. Then each worker computes in parallel the stencil step on the local object set. This basically leads to a kind of BSP implementation with communication and computation phases alternated. Communication grouping may be performed such that the computation phase latency is optimized.

Suitable implementations of the stencil pattern defined above may take into account the “structure” of the stencil.

A notable example takes into account cache management issues. Suppose that our domain X is a matrix, and the stencil function $f_N$ always returns elements to the left, right, above and below the current element\(^2\) (i.e. $F_N(X_{i,j}) = \{X_{i-1,j}, X_{i+1,j}, X_{i,j-1}, X_{i,j+1}\}$). Now, when computing $f_F(F_N(X_{i,j}))$, access to the neighbouring objects $X_{i-1,j}, X_{i+1,j}$ happens (probably) in the same cache line, while access to objects $X_{i,j-1}, X_{i,j+1}$ would in general require accessing different cache lines. If we mark the stencil items with colours (e.g. $X_{i-1,j}, X_{i+1,j}$ as white and $X_{i,j-1}, X_{i,j+1}$ as black), and subsequently rearrange the third “map reduce” phase in such a way that all the white neighbours contributions are computed first and the black ones are computed later, we should observe a certain speedup. Recent results on stencil index rearrangement have demonstrated that this kind of optimization may be very efficient.

A further variation of the stencil pattern is highly relevant for one of the use cases as described in D6.2. For some stencils, the results of $f_F$ are relevant not only for $X_i$, but also for $X_j$. Typically, $f_F(X_i, X_j) = -f_F(X_j, X_i)$, as for instance in the case of forces in molecular dynamics and fluxes across boundaries for FVM. Thus, in order not to calculate the same interaction twice, the result is written both to $X_i$ and $X_j$. Obviously, this may lead to race conditions when two parallel threads write to the same object. One way to avoid such race conditions is colouring. Only objects of the same colour can be run in parallel. Objects $X_1$ and $X_2$ are of the same colour, if the intersection of $(X_1, F_N(X_1))$ and $(X_2, F_N(X_2))$ is empty.

In both cases, colours used to implement cache optimization or to guarantee mutual exclusion, the $f_N$ function has type $f_N : X \to \{(X, C)\}$ where $C$ is the set of supported colours.

\(^{2}\)we are not taking into account borders, for the moment
Chapter 3

Conclusions

This deliverable was intended to summarize the different application (domain) specific parallel patterns used within the project. The interaction among project beneficiaries led to the list of patterns of Tab. 1.1. Among these patterns we identified a set of “representative” patterns as those most likely to be candidates for implementation during WP2 activities eventually leading to D2.6 (at M24). These patterns have been described within this document according to state-of-the-art design pattern definition standards, that is by describing pattern, context, forces and solutions as in [11].

We expect that D2.6 will contain an implementation of most of the patterns detailed in this report, although the interaction between WP2 and WP6 beneficiaries is expected to result in “variants” of these representative patterns more suitable for use in the implementation of the use cases than the “general” patterns described here.

This deliverable assesses two main results:

- The “structure” of application specific patterns differs from application domain to application domain. For example, in symbolic computation patterns are usually represented as higher order functions with both functional and non-functional parameters, whereas in soft computing, patterns are often described in an algorithmic way. Also, within a given domain different patterns may be expressed in different ways. In the numerical computation domain we identified a pattern which basically is specified through a kind of imperative (pseudo-)code and a pattern following again the “higher order function” specification more typical of the symbolic computation patterns. This poses challenges for WP3 refactoring techniques supporting pattern introduction in sequential code, but does not change the way patterns are described (and eventually implemented through suitable compositions of algorithmic skeletons) in WP2.
• The domain specific patterns may be implemented and optimized using compositions of core patterns—that in turn correspond to compositions of RISC skeletons—such as those described in D2.1: pipelines, task farms, maps and reduces, etc. We expect the implementation of the high level domain specific patterns with the RISC skeletons discussed in D2.2 may possibly lead to the identification of new features to be included in the skeleton implementation, although we do not expect that these new features will radically change the RISC skeleton set composition and internal features.
Bibliography


