Project no. 644235

REPHRASE

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

Report on shaping and pattern discovery for advanced patterns D2.9

Due date of deliverable: M33

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

Responsible institution: UC3M
Editor and editor's address: Manuel F. Dolz, David del Rio Astorga, J. Daniel García, UC3M

Version 0.1

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

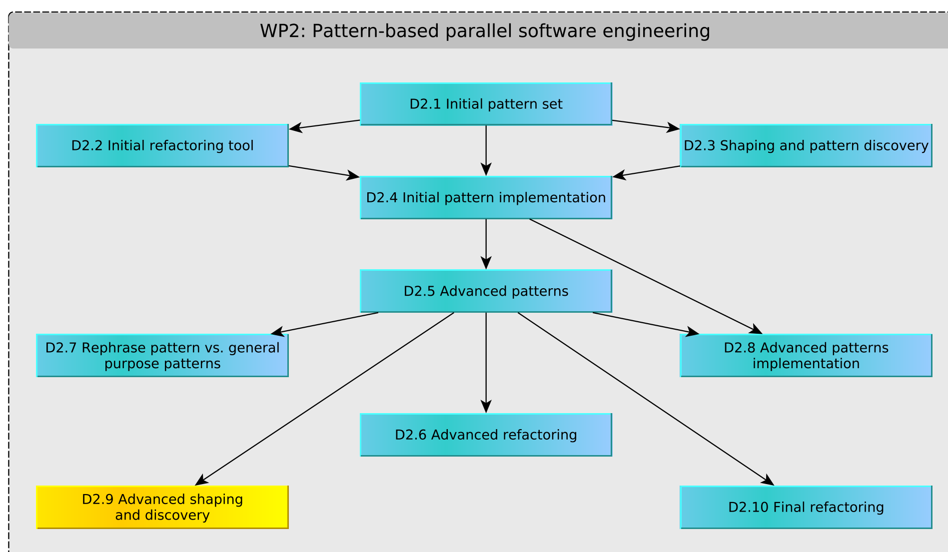
Rev.	Date	Who	Site	What
1	15/11/17	Manuel F. Dolz	UC3M	Submitted first version/PPAT tool
2	22/12/17	Evgueni Kolossov	PRL	Advanced program shaping techinques
3	29/12/17	Marco Danelutto	UNIPI	Minor fixes
4	31/12/17	Manuel F. Dolz	UC3M	Added chapter for software references

Executive Summary

This document is the ninth deliverable from WP2 “Pattern-Based Parallel Software Engineering”. It provides the following contributions *i)* a coherent, complete and formalised set of parallel patterns for data-intensive applications, together with a domain specific language (DSL) to represent them; *ii)* pattern implementations on top of existing parallel programming frameworks; *iii)* a pattern discovery methodology (for widely used legacy and existing parallel applications); *iv)* C++ program shaping/componentisation techniques; and *v)* refactoring rules and tool-support for the introduction and tuning of parallel patterns in both new and existing C++ applications.

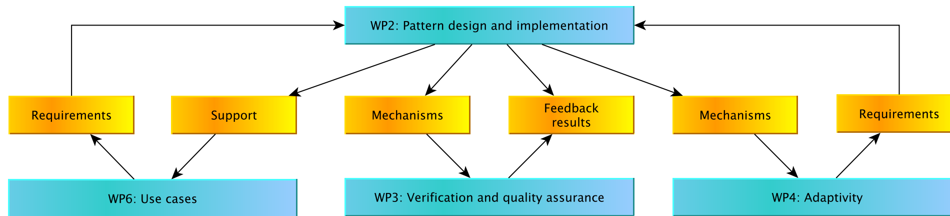
This deliverable, D2.9 “Report on program shaping and pattern discovery for advanced patterns.", integrates the results of the different phases of WP2 (T2.3 “Program Shaping” and T2.4 “Pattern Discovery”) where, according to the DoW *we will extend the program shaping techniques to the advanced pattern set* and *describe the advanced pattern discovery techniques* for the advanced pattern set. The work in this deliverable is divided in two steps. First, it defines the program shaping techniques and methods for the advanced pattern set of **RePhrase**. These techniques allow refactoring of sequential C++ programs into hygienic C++ code with equivalent functionality by eliminating non-hygienic code properties, such as side-effects and unnecessary task/data dependencies. The second step holds two sub-steps: it defines pattern candidates along with their conditions and properties to be introduced in C++ applications; and extends our prototype for pattern discovery to identify instances of advanced parallel pattern candidates in C++ applications.

The placement of D2.9 in the WP2 overall deliverable list is summarized by the following schema:



while the strict influences between pattern design and implementation in WP2 and

activities in the other major technical workpackages are summarized by the following schema:



The contributions per partner of this deliverable led by UC3M are the following:

- PRL has contributed in Chapter 2, reviewing advanced program shaping techniques and methods in order to refactor sequential C++ programs into hygienic C++ code with equivalent functionality.
- UNIPI and UC3M have contributed in Chapter 3, describing for each pattern of the advanced set of **RePhrase** the conditions and requirements that need to be satisfied in order to be introduced in C++ applications.
- UC3M has contributed in Chapter 4, presenting a prototype pattern discovery tool that, using the aforementioned conditions and a static approach, identifies instances of some advanced patterns in C++ applications at compile time.
- UC3M has produced Chapter 5 referencing the repositories where the software presented in this deliverable can be found.
- UC3M has produced Chapter 6 enumerates a few concluding remarks and future works.

Contents

Executive Summary 2

1 Introduction **6**

2 Program shaping methods and techniques **8**
2.1 Introduction 8
2.2 *pure* Functions 8
2.3 `constexpr` Functions in C++'17 9
2.3.1 *Literal* Types in C++'17 9
2.3.2 *Core Constant Expression* in C++'17 10
2.4 Relationship Between *pure* and `constexpr` Functions 10
2.5 Declaring Functions and Functors as `constexpr` where possible 11

3 Advanced pattern discovery techniques **13**
3.1 Pool pattern 13
3.2 Image Convolution pattern 14
3.3 Stream iteration pattern with multiple outputs 15
3.4 DASP patterns 15
3.4.1 Windowed stream farm 15
3.4.2 Keyed stream farm 16

4 Parallel pattern discovery tool **18**
4.1 Related work 18
4.2 Abstract Syntax Trees for Pattern Discovery 19
4.3 Parallel pattern detection 20
4.3.1 REPHRASE attributes for parallel patterns 20
4.3.2 Parallel Pattern Analyzer Tool 21
4.3.3 Pool detection module 22
4.3.4 Image Convolution detection module 23
4.3.5 Window Farm detection module 24
4.4 Evaluation 25
4.4.1 Detection of the Pool pattern 25
4.4.2 Detection of the Image Convolution pattern 25
4.4.3 Detection of the Window Farm pattern 27

5 Software availability **29**

6 Conclusions and future works **30**

1. Introduction

Parallel design patterns have been recognized since a long time viable tools to overcome the problem of rapid and efficient development of parallel applications [3, 12] and the **RePhrase** project has been designed to exploit the potential of parallel pattern in the data intensive application scenario. If properly tuned, they are good approaches in order to efficiently exploit parallel computing architectures [6, 22]. Pattern-based programming models that encapsulate algorithmic aspects using a building blocks approach [13]. Fundamentally, parallel patterns offer a way to implement robust, readable and portable solutions, while hide away the complexity behind concurrency mechanisms, e.g., thread management, synchronizations or data sharing. Numerous examples of pattern-based programming frameworks, such as SkePU [7], FastFlow [2] or Intel TBB [18], can be found in the literature.

One of the main aims of the WP2 from the **RePhrase** project is to provide the application programmers a comprehensive set of parallel patterns that may be used to implement efficient parallel applications. During the **RePhrase** project, the individual study of the use cases in WP6 have revealed the need for a set "high-level" patterns to support their data-intensive computations and the algorithmic structures. The set of "high-level" patterns was defined in D2.5 extends with advanced patterns the initial set presented in D2.1. Concretely, we refer to "advanced" patterns those designed for scenarios in which the basic patterns do not match any of these constructions or have to be composed in a very complex way. This occurs in many domain-specific algorithms coming from the evolutionary and symbolic computing [8] domain, wireless sensor networks algorithms [5] or real-time processing engines [17].

Continuing with the **RePhrase** philosophy, this deliverable includes the set of advanced program shaping techniques and methods in order to refactor sequential C++ programs into hygienic C++ code with equivalent functionality by eliminating non-hygienic code properties, such as side-effects and unnecessary task/data dependencies. This document also enumerates the set of requirements that these advanced patterns have to met in order to detect potential candidates in sequential codes. Next, the deliverable introduces the extension made in the Parallel Pattern Analyzer Tool (PPAT) to support new modules able to encounter some of the advanced patterns. For each advanced pattern, the detection process is illustrated with a worked example. In general, all these facts motivate the contributions presented

in this deliverable, namely D2.9, from the **RePhrase** project. The deliverable contents are organized into three main parts:

- Chapter 2 reviews program shaping techniques and methods in order to refactor sequential C++ programs into hygienic C++ code with equivalent functionality.
- Chapter 3 describes for each pattern of the advanced set of **RePhrase** the conditions and requirements that need to be satisfied in order to be introduced in C++ applications.
- Chapter 4 extends our prototype pattern discovery tool for detecting some of the patterns in the **RePhrase** "advanced" set of patterns. This tool uses some of the requirements defined previously to encounter potential candidates of those patterns.
- Chapter 5 we detail where the software discussed in this deliverable can be found.
- Chapter 6 enumerates a few concluding remarks and future works. discussed in the deliverable may be found.

2. Program shaping methods and techniques

2.1 Introduction

All of the program shaping techniques identified in D2.3 for the initial pattern set described in D2.1 also apply to the advanced pattern set described in D2.5. We will therefore refine and expand the program shaping techniques in the following sections.

This part of the document expands on the concept of *hygienic* properties of a function and compares those properties with the requirements for `constexpr` functions and *core constant expressions* in C++'17. It shows how the application of the `constexpr` specifier to functions helps in writing *hygienic* code to facilitate the application of parallelisation patterns.

2.2 *pure* Functions

In D2.3 we described the properties a function must hold for it to be considered *pure*. To recap, a function will be non-*pure* if it reads or writes to an object other than:

- a non-volatile automatic variable, or
- a function parameter, or
- an object allocated within the body of the function.

Also, a function will not be considered as pure when:

- The body contains loops which can be statically determined as infinite, or
- The function can be statically determined as directly or indirectly recursive
- The function calls a function that causes the program to exit immediately, for example:
  - `std::exit`

  - `std::abort`
  - `std::terminate`

- The function calls `std::longjmp`

2.3 `constexpr` Functions in C++'17

In C++'11 the concept of a C++'03 *constant integral expression* was generalised to a *core constant expression*. It also introduces *literal types* and a `constexpr` specifier for functions and objects. While `constexpr` functions were initially restricted to exactly one `return` statement in C++'11, this was relaxed for C++'17 and most statements (including iteration statements) are now allowed to appear in a `constexpr` function.

In C++'17 a `constexpr` function shall:

- not be `virtual`
- have a *literal* type as its return type
- have only parameters of *literal* type
- not have an *asm-definition*, a `goto` statement, an identifier label, or a *try-block* in its function body
- not define a variable of non-*literal* type or of static or thread storage duration or for which no initialisation is performed

As an example, the following sample code does not compile (`-std=c++17`) due to the `static` variable declaration:

```
constexpr int foo(int i)
{
  // object with static storage duration
  // not allowed in constexpr function
  static int s = 0;
  s += i;
  return i;
}
```

2.3.1 *Literal* Types in C++'17

A *literal* type is a type that is:

- `void`, scalar type, reference type, or an array of literal type
- a class type with a trivial destructor, at least one `constexpr` constructor or constructor template, and all of its non-static data members and base classes of non-volatile literal type

As an example, the following code defines a legal literal type:

```
// A is a literal type
struct A
{
  constexpr A(int i)
    : m_i(i)
  { }

  constexpr int get() const
  {
    return m_i;
  }

 private:
  int m_i;
};
```

2.3.2 *Core Constant Expression* in C++'17

Simply put, a *core constant expression* is an expression that can be evaluated at compile time. One of the requirements is that any such evaluation must not trigger undefined behaviour or exceed implementation defined limits.

```
constexpr int foo(int i)
{
  return 100 / i;
}

// 'foo(0)' not a constant expression as
// it would trigger undefined behaviour
static_assert(foo(0) == 0, "failed");
```

By definition, such an evaluation can't have any side-effects and is also *hygienic*.

Note that although only `constexpr` functions can be called from *core constant expressions*, not all calls of `constexpr` functions are necessarily *core constant expressions* as it depends on the parameter values whether the evaluation of the function body is a *core constant expression*—a non-constant sub-expression doesn't affect the *constness* if it is not evaluated itself.

2.4 Relationship Between *pure* and `constexpr` Functions

Although the concept of a *pure* function is very similar to a `constexpr` function in C++, it's not a simple one-to-one relationship between the two.

While some of the constructs that are considered *unhygienic* are also forbidden in `constexpr` functions, like the definition of variables with static storage duration, other *unhygienic* properties like the modification of global objects are not generally forbidden in `constexpr` functions—these only result in an evaluation not being a *core constant expression*.

```
int g = 0;

// constexpr function that is not pure
constexpr int foo (int i, bool b)
{
  if (b)
  {
    g += i;
  }

  return i;
}
```

A constexpr function therefore doesn't imply that it is also *pure*.

On the other hand, not all *pure* functions that are considered *hygienic* can be written as constexpr functions, as any use of non-*literal* types or a call to a non-constexpr function is not allowed in a constexpr function.

```
// pure function that can't be made constexpr
int foo (int i)
{
  // new is not constexpr, but allowed in a pure function
  int *ptr = new int(i);
  int j = *ptr;
  delete ptr;
  return j;
}
```

So being *pure* doesn't imply constexpr either. But a constexpr function that results in a *core constant expression* when being called with any arguments, is a *hygienic* function.

2.5 Declaring Functions and Functors as constexpr where possible

Despite the differences shown, it is good practice to declare any function or functor used in algorithms as constexpr where possible as it prevents the use of some *unhygienic* properties and documents that the function can be used without side-effects in some contexts.

As the constexpr function won't be called with constant expressions as arguments when being used in an algorithm, the function will still be evaluated at runtime as usual. But if a call to the function results in a *core constant expression* for the range of all possible input values, the function is automatically *hygienic*.

```
#include <algorithm>
#include <vector>

// hygienic constexpr function
constexpr unsigned char foo (unsigned char i)
{
  return i % 16;
}
```

```
void bar(std::vector<unsigned char> const & i)
{
  std::vector<unsigned char> o;
  o.resize (i.size ());

  std::transform (i.cbegin (), i.cend (), o.begin (), foo);
}
```

In some cases where the range of possible input values is limited, it can be asserted at compile time that each of these input values will result in a *core constant expression* when the function is called and that therefore the function is in fact *hygienic*.

For example, the following `static_assert` ensures that `foo` called with arguments in the range $[0, 256)$ will always result in a *core constant expression*, so the function will be *hygienic* for those input values:

```
static_assert ( ([] () {
  for (unsigned int i = 0; i != 256u; ++i) {
    foo (i);
  }
} (), true));
```

3. Advanced pattern discovery techniques

This part of the document enumerates the properties and conditions for each of the patterns of advanced set from the **RePhrase** project, that need to be satisfied in order to be applied into a C++ application. Some of these requirements will be used for the Parallel Pattern Analyzer tool presented as a prototype in the next chapter of this document. We follow the same classification of patterns listed in deliverable D2.5 for describing their requirements. To ease the understanding, we reference each pattern described with references to D2.5. We refer these parallel design patterns "advanced", designed for those scenarios in which the basic patterns do not match any of these constructions or have to be composed in a very complex way. This occurs in many domain-specific algorithms coming from the evolutionary and symbolic computing [8] domain, wireless sensor networks algorithms [5] or real-time processing engines [17]. Therefore, we detect a clear need for having advanced patterns in order to simplify the development of sophisticated algorithms related to the aforementioned application domains.

In the following sections we describe the conditions that have to met in order to find candidates of the advanced "high-level" patterns: Pool, Image Convolution, stream iteration pattern with multiple outputs and DASP patterns.

3.1 Pool pattern

The Pool pattern models the evolution of a population of individuals. Iteratively, selected individuals are subject to evolution steps. The resulting new individuals are inserted in the population or discarded according to their "fitness" score. The process is iterated up to a given number of iterations (or up to a given computation time) or up to the point an individual with a given fitness is inserted in the population. Low fitness individuals may be removed from the population to keep the population size constant at each iteration. More details about this pattern can be found in Deliverable 2.5, Section 2.1.

Requirements The requirements of this pattern should determine whether a loop can be refactored into a Pool pattern. These requirements are described as follows:

- *Multiple stages*. The loop should be able to be divided in more than one stage, corresponding to the selection, evolution, filtering and termination functions.
- *Interconnected stages*. Each stage in the Pool should receive, as inputs, the outputs from the previous stage.
- *Pure evolution function*. The stage corresponding to the evolution function should be pure, i.e., it can be computed in parallel with no side effects.
- *No global variables modified*. There should not exist instructions that modify global variables in the main loop.
- *Feedback loop iterations*. There should exist feedback in the population the filtering or selection stage. This requirement ensures progress on the Pool pattern, as the population has to be modified in each iteration.
- *No break statements*. There should not exist break statements (`continue`, `break` or `return`) in the loop, as they cannot be paralelized. However, they can be allowed in inner scopes of the main loop.

3.2 Image Convolution pattern

This pattern computes the image convolution according to some input "kernel" parameter. A kernel parameter is an $N \times N$ matrix of integer values. The image convolution is obtained from the source image processing each pixel at position i, j by taking the $N \times N$ values centered at i, j, multiplying each of the values by the corresponding value of the kernel and summing up all the results to get the new i, j pixel of the resulting matrix. Basically, the Image Convolution pattern is clearly a specialization of the stencil pattern. More details about this pattern can be found in Deliverable 2.5, Section 2.2.

Requirements The requirements of this pattern should determine whether a loop can be refactored into a parallel Image Convolution pattern is, i.e., if its iterations are independent each other and if there are neighborhood relationships between the input and output data sets. This operation also requires the presence of a kernel. These requirements are described as follows:

- *Neighborhood dependencies*: Each element of the output data set should depend on, at least, one element of the input data set and its neighbors.
- *No global variables modified*. There should not exist instructions that modify global variables in the loop.
- *No RAW dependencies*. There should not exist RAW dependencies of variables used within iterations of the loop.

- *No break statements*. There should not exist break statements (`continue`, `break` or `return`) in the loop, as they cannot be paralelized. However, they can be allowed in inner scopes of the main loop.
- *Presence of a kernel*: Each operation performed on the element i, j and its neighbors should be performed with a kernel, normally a matrix of $N \times N$ integer values.

3.3 Stream iteration pattern with multiple outputs

The stream iteration pattern in D2.1 assumed that a computation is repeatedly applied to a single stream item up to the point a given condition holded true. At the end, the result of the computation was delivered onto the output stream. The stream iteration pattern with multiple outputs, instead, assumes that at each iteration a value may be output onto the output stream-depending on the value of a "output guard" function such that the cardinality of the stream may vary. More details about this pattern can be found in Deliverable 2.5, Section 2.3.

Requirements The requirements of the stream iteration pattern with multiple outputs can be summarized as follows:

- *Computing function*. The computing function should be a pure function that, applied to an input stream element, produce another element of the same type.
- *Equal input/output stream types*. The input and output stream data type should be the same.
- *Nested loop*. There should be an inner loop in this portion of code that contains the computing functions. The number of iterations of this loop is given by the results of a boolean predicate.
- *Output guard function*. There should exist a boolean predicate that determines when an item, processed through each iteration of the nested loop, has to be delivered to the output stream.
- *Termination function*. There should exist a boolean predicate that determines the number of iterations of the nested loop. This function depends on input stream data type.

3.4 DASP patterns

3.4.1 Windowed stream farm

The windowed stream farm pattern computes a function on "windows" of stream item values. Concretely, this pattern implements a computation that outputs items

on the output stream corresponding to the evaluation of a given function over successive, consecutive windows of items appearing on the input stream. The "windows" have a length (number of items to be listed in the window) and an overlap factor (number of items in window w_i also appearing in window w_{i+1}). More details about this pattern can be found in Deliverable 2.5, Section 2.4.1.

Requirements The requirements of this pattern should determine whether a loop can be refactored into a windowed stream farm pattern. These requirements are described as follows:

- *Multiple stages*. The loop should be able to be divided in more than one stage, corresponding to the window generation, kernel stage and window update.
- *Interconnected stages*. The found loop stages should receive, as inputs, the outputs from the previous stage.
- *Pure kernel function*. The stage corresponding to the kernel function should be pure, i.e., it can be computed in parallel with no side effects.
- *No global variables modified*. There should not exist instructions that modify global variables in the main loop.
- *Feedback loop iterations*. There should exist feedback in the window update and window generation stages. This is because the window, represented as a buffer, should be updated in each iteration of the pattern.
- *Item accumulation*. The window generation function should perform an accumulation operation in order to conform the window. In other words, each time that a new item arrives from the input stream, this should be appended at the end of the buffer until the window is finally conformed.
- *No break statements*. There should not exist break statements (`continue`, `break` or `return`) in the loop, as they cannot be paralelized. However, they can be allowed in inner scopes of the main loop.

3.4.2 Keyed stream farm

The keyed stream farm computes a function on "windows" of stream item values. Each input item belongs to a unique class called key, i.e. in other words the physical stream can be viewed as a multiplexing a several logical streams, each one conveys the items with the same key value. This pattern implements a computation that outputs items on the output stream corresponding to the evaluation of a given function over successive, consecutive windows of items appearing on the same logical input stream. More details about this pattern can be found in Deliverable 2.5, Section 2.4.2.

Requirements The requirements of this pattern should determine whether a loop can be refactored into a keyed stream farm pattern. These requirements are described as follows:

- *Multiple stages*. The loop should be able to be divided in more than one stage, corresponding to the window generation, kernel stage and window update.
- *Interconnected stages*. The found loop stages should receive, as inputs, the outputs from the previous stage.
- *Pure kernel function*. The stage corresponding to the kernel function should be pure, i.e., it can be computed in parallel with no side effects.
- *No global variables modified*. There should not exist instructions that modify global variables in the main loop.
- *Feedback loop iterations*. There should exist feedback in the window update and window generation stages. This is because the window, represented as a buffer, should be updated in each iteration of the pattern.
- *Item accumulation*. The window generation function should perform an accumulation operation in order to conform the window. In other words, each time that a new item arrives from the input stream, this should be appended at the end of the buffer until the window is finally conformed.
- *Keyed windowing*. As the keyed farm assumes that the items in the input stream are basically "key-value" tuples, the accumulation operation has to conform a window depending on the input item key. In this case, as the pattern has to handle multiple windows, one per key, the input item key determines to which of these windows the item value has to be appended.
- *No break statements*. There should not exist break statements (`continue`, `break` or `return`) in the loop, as they cannot be paralelized. However, they can be allowed in inner scopes of the main loop.

4. Parallel pattern discovery tool

In this part of the document we extend the Parallel Pattern Analyzer Tool (PPAT) as a prototype tool to detect potential advanced parallel patterns of the **RePhrase** project in sequential applications. The features of this tool are the following:

i) it is completely independent of the refactoring tool used, since it identifies parallel patterns;

ii) it performs a static analysis and avoids the use of profiling techniques, thus becomes much faster than other approaches; and;

iii) it gives hints to users when a code region complies with the requirements of a given advanced parallel pattern.

4.1 Related work

We find several research works in the state-of-the-art that address the detection of potential parallel codes and refactoring processes. However, the detection task is not simple and the tools developed to identify parallel patterns in sequential codes are strongly tied to the programming language requiring, most of them, profiling techniques. For example, the approach developed by Sean Rul et al. [21] leverages LLVM to instrument loops in the sequential code and performs an LLVM-IR profiling analysis to decide whether a loop is a pipeline or not. After that, it transforms the code to produce a parallel source code. However, this tool presents some shortcomings: it needs to execute the target application several times and profile it. Also, it is tied to the C programming language. Our approach addresses these limitations by means of performing a static analysis without requiring any previous execution or profiling techniques, and supports both C/C++ programming languages.

Other contributions, such as the work by Molitorisz et al. [15], detect statically potential parallel patterns, nevertheless they do not check for dependencies, so the correctness of the resulting parallel application cannot be guaranteed. Instead, they need a subsequent execution to discover potential data races and dependencies. Our work addresses this issue by checking memory accesses statically, i.e., at compile-time. Likewise, PoCC [1], a flexible source-to-source compiler using polyhedral compilation, is able to detect and parallelize loops, however it does not take into

account high-level parallel patterns. On the other hand, we find tools that detect parallel patterns using only profiling techniques. For example, DiscoPoP in [11], leverages dependency graphs in order to detect parallel patterns. Nevertheless, this tool has an important drawback: the profiling techniques have a non-negligible execution time and memory usage. A similar approach, presented by Tournavitis et al. [23], detects and transforms sequential code into parallel introducing parallel pipeline patterns. Alternatively, FreshBreeze [10], a dataflow-based execution and programming model and computer architecture, leverages static loop detection techniques that analyze dependencies and transform parallelizable loops using a task tree-structured memory model.

It is important to remark that, approaches based on static analysis are not very extended in the area, since analyzing data dependencies becomes much more complex at compile time. Orthogonally, some works take advantage of functional languages. For instance, István Bozó et al. [4] develop a tool that detects parallel patterns in applications written in Erlang. Compared to other languages, Erlang features make the detection process much simpler. Nonetheless, the tool requires profiling techniques in order to decide which pattern suits best for a concrete problem.

The parallel pattern detector prototype presented in this report differentiates from the previous examples in different aspects: *i)* it leverages a static analysis without requiring any previous execution or profiling techniques, *iii)* it checks dependencies due to memory accesses statically, i.e., at compile-time, *ii)* it supports both C and C++ programming languages, *iii)* it follows a modular design that allows us to easily extend its functionality for detecting pattern.

4.2 Abstract Syntax Trees for Pattern Discovery

In this section we describe the Abstract Syntax Tree (AST), as a syntactic structure representation of the source code in a tree model and normally generated by the compilers at compile time [16]. These trees contain information about variables, operators and functions used in the source code. Figure 4.1 shows an example of source code along with its associated AST. In this work, we leverage the Clang compiler library to develop a tool that uses its AST to analyze code statically and determine whether a loop can be represented as a pipeline pattern.

We describe next the definitions of the kind of references in order to clarify concepts used in the successive sections. A basic interpretation of these kinds, based on the C++11 standard [14], are the following:

- Lvalue: It usually appears on the left-hand side of an assignment expression and designates a function or an object, being addressable but not assignable.
- Rvalue: It might appear on the right-hand side of an assignment expression and defines a temporary (sub)object or a literal value, thus being non-addressable but assignable.

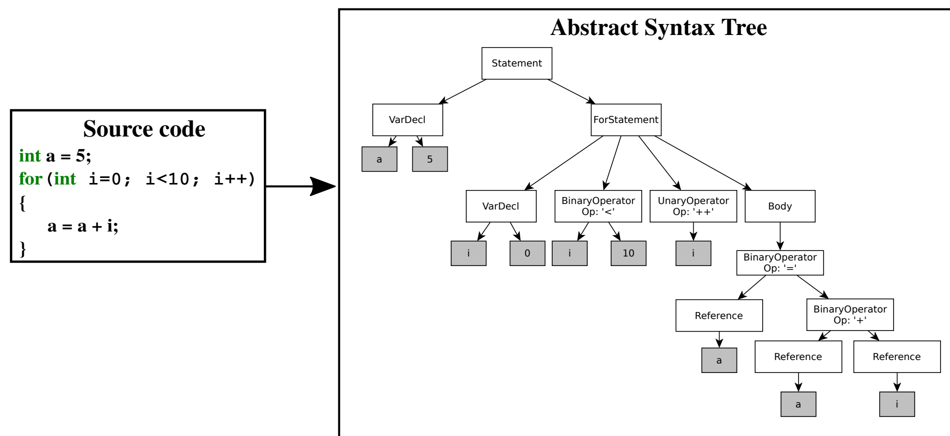


Figure 4.1: Example of Abstract Syntax Tree for a given source code.

In this work, we have redefined these concepts in order to reflect the use of references in the code. Therefore, we refer to Rvalue as a read-only reference, while Lvalue designates any modifiable value. In next sections, we explain the use of these definitions within the Parallel Pattern Analyzer Tool.

4.3 Parallel pattern detection

In this section we revisit the Parallel Pattern Analyzer Tool (PPAT), firstly presented in D2.3. The implementation of this tool leverages the Clang library to generate the AST and walk through it in order to collect relevant information about the source code. Afterwards, a series of modules checks the set of requirements set in the previous chapter for the different parallel patterns in order to annotate the code that can be refactored into parallel patterns. In this case, we present some examples to detect the Pool, Image Convolution and Window Farm patterns.

4.3.1 REPHRASE attributes for parallel patterns

In order to annotate parallel patterns using custom C++11 attributes [9], we have extended the set of attributes defined for the projects REPARA [19] and REPHRASE [20]. Table 4.1 describes the attributes used for annotating the Pool, Image Convolution and Window Farm patterns.

Thanks to these attributes, a refactorization tool would have enough information to transform annotated code regions into parallel. Also, it allows us to completely separate the detection and refactorization processes, so that, different tools can be used for the last step. Particularly, in this deliverable we only pursue the detection phase.

It is worth noting that in D2.3 the tool was established to find some of the patterns in the "initial set". Given the simplicity of such patterns, the tool was able to ensure the detection correctness. Unfortunately, for the "advanced" pattern

Table 4.1: REPHRASE attributes for advanced patterns.

REPHRASE Attribute	Description
`rph::pool`	It identifies a Pool pattern.
`rph::poolid`	It is related to `rph::stage` and includes the Pool ID.
`rph::window_farm`	It identifies a Window Farm pattern.
`rph::wfarmid`	It is related to `rph::stage` and includes the Window Farm ID.
`rph::stencil`	It identifies a Stencil pattern.
`rph::convolution`	It identifies a Convolution pattern.
`rph::stage`	It identifies a code section as a stage.
`rph::in`	This attribute references the input variables of a pattern.
`rph::out`	It references the pattern output variables.

set, the detection correctness cannot be guaranteed. This is mainly given by the inherent difficulties and different features that these pattern have to met in order to fully ensuring that they have been encountered. Therefore, the annotations inserted by PPAT for the advanced patterns should be taken just as hints.

4.3.2 Parallel Pattern Analyzer Tool

In this section, we review the Parallel Pattern Analyzer Tool (PPAT). This tool takes advantage of the Clang library to generate the Abstract Syntax Tree (AST) and walk through it in order to collect relevant information about the source code and identify parallel patterns.

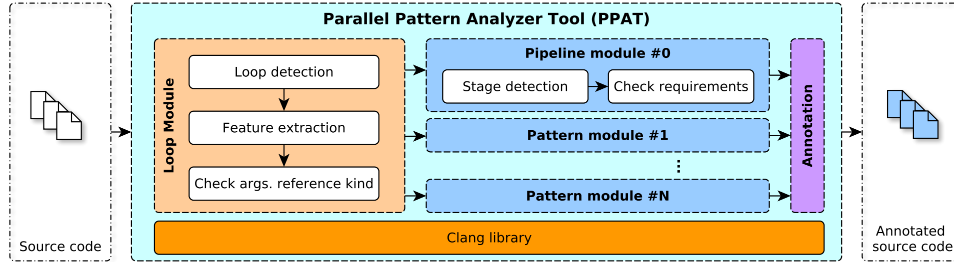


Figure 4.2: Workflow diagram of PPAT.

Figure 4.2 depicts the general workflow diagram of PPAT. First, the tool receives the sequential source code files that should be analyzed. Next, the following steps are executed:

1. *Loop detection*. This step detects potential loops that can be transformed into parallel patterns. Basically, it iterates the AST, extracts loop-related subtrees and gathers information of different AST nodes (e.g. variables, function calls, conditional statements). On the other hand, it collects information about the functions implemented in the source code.

2. *Feature extraction.* This step leverages the structures collected in the previous step in order to extract specific features about variable declarations, references, function calls, inner loops, memory accesses, operations, etc. Next, for each statement encountered, we store information about location on the original code, variable and functions name, reference kind (Write or Read) and global storage references.

3. *Check arguments reference kind.* The last step checks whether the kinds of variable references passed as arguments in functions can be determined or not. In some cases, it is not possible to know statically if the kind of arguments passed by reference are read or written.

Finally, marked loops are passed to the different pattern analyzer modules. In the last stage, parallel pattern modules annotate loops, using REPHRASE attributes, that can be refactored into parallel codes. The following sections describe the new modules for detecting some of the advanced pattern proposed in the **RePhrase**

4.3.3 Pool detection module

This section describes the internal workings of the Pool module responsible for checking the set of requirements stated for the Pool pattern. These requirements are controlled using the following constraints:

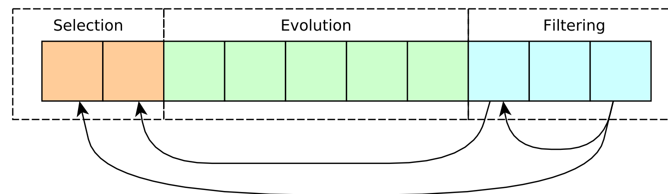


Figure 4.3: Stages of the Pool pattern.

1. *Stage detection.* This step is responsible for identifying potential stages in which a loop can be split. Our strategy creates a new stage each time a function call or an inner loop is found in the main loop. Afterwards, for each stage encountered, it checks whether the stage is fed with, at least, one previous stage output. If this requirement is not met, the complete stage is merged with the previous one until all stages comply with this requirement.

2. *Feedback detection.* In this step, the tool detects all potential feedbacks among stages, i.e., variables that are read in the stage S_i and written in stage S_{i+n}.

3. *Stage classification.* The goal of this step is to classify the stages found in the previous step into the three Pool phases:

- *Selection function.* In this step the tool determines which stages belong to the selection function using the feedbacks detected in the previous step. Note that the selection function may be composed of multiple consecutive stages. As can be seen in Figure 4.3, the selection function involves the first s stages (see stages highlighted in orange). The last stage conforming this function is the last with feedback from a coming stage.
- *Filtering function.* To detect which stages conform the filtering function, the tool inspects what is the stage in the loop producing feedback to the selection function. With this, the filtering function is determined by the collection of consecutive stages starting from the previously identified stage till the last in the loop (see stages marked in blue).
- *Evolution function.* The classification of stages for determining the evolution function is straightforward. In this case the tool takes all remaining stages (placed in middle positions) to identify the evolution function (see stages highlighted in green).

It is important to remark that the stages sets determined to conform each of the Pool functions should not be empty. Otherwise, the loop cannot be classified as a Pool pattern.

4.3.4 Image Convolution detection module

In this section we describe in detail how the PPAT module for detecting the Image Convolution pattern works. Note that the detection of this pattern is very similar to the Stencil pattern. These requirements are controlled using the following constraints:

- *Known number of input elements.* The input data must be declared and allocated before the definition of the analyzed loop.

- *At least one output.* According to the previous Image Convolution definition, the kernel function of this pattern processes an input to produce an output. So, the set of outputs for a given loop should not empty.

- *Neighborhood.* Each output item should depend on more than one element from the input data collection. To check this requirement, the tool inspects the indexes used to access the potential neighborhood. Thus, this requirement is met if there exist accesses to multiple input items in a same loop iteration.

- *Kernel.* This requirement checks if each coordinate in the image i, j uses a $N \times N$ kernel. This requirement is only met if there exist a reduction operation inside the main loop that walks across the aforementioned convolution kernel and its related neighborhood.

4.3.5 Window Farm detection module

This section describes the internal workings of the Window Farm module responsible for checking the set of requirements stated for the Window Farm pattern. These requirements are controlled using the following constraints:

1. *Stage detection.* Similar to the stage detection in the Pool pattern, the Window Farm PPAT module also uses the same strategy for splitting the loop into stages. Basically, this strategy creates a new stage each time a function call or an inner loop is found in the main loop. Afterwards, for each stage encountered, it checks whether the stage is fed with, at least, one previous stage output.

2. *Feedback detection.* In this step, the tool detects all potential feedbacks among stages, i.e., variables that are read in the stage S_i and written in stage S_{i+n}.

3. *Stage classification.* The goal of this step is to classify the stages found in the previous step into the three Window Farm phases:

   - *Window generation.* In this step the tool determines which stages belong to the window generation function using the feedbacks detected in the previous step. Note that the window generation may be composed of multiple consecutive stages. Similar to the Pool pattern, this generation involves the first s stages. The last stage conforming this function is the last with feedback from a coming stage.

     Apart from the previous requirements, PPAT also checks that all the operations performed in these stages on the variables marked with "feedback", are reduction operations, e.g., `vector.push_back(i)`.
   - *Window update.* To detect which stages conform the window update function, the tool inspects what is the stage in the loop producing feedback to the window generation function. With this, the window update is determined by the collection of consecutive stages starting from the previously identified stage till the last in the loop.
   - *Kernel function.* The classification of stages for determining the kernel function is straightforward. In this case the tool takes all remaining stages (placed in middle positions) to identify the evolution function. This behavior is similar to the Pool pattern.

     Additionally, the kernel function needs to be a pure function in order to determine that the code snipped corresponds with a Window Farm pattern.

   It is important to remark that the stages sets determined to conform each of the Window Farm functions should not be empty. Otherwise, the loop will not be classified as a potential Window Farm pattern.

4.4 Evaluation

In this section we perform an experimental evaluation of PPAT using simple benchmarks that analyze how the advanced patterns can be detected. To do so, we have used the following hardware and software components:

- *Target platform*. The evaluation has been carried out on a server platform comprised of 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.40 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM, while running Linux Ubuntu 14.04.2 LTS OS on a 3.13.0-57 Linux kernel.
- *Software*. The compilation of PPAT has been performed using the LLVM compiler infrastructure v3.7.0, with its Clang compiler and the extended attributes from RePhrase.

Our evaluation methodology proceeds as follows. We have designed three synthetic benchmarks containing the Pool, Image Convolution and Window Farm patterns. Afterwards, we have used the PPAT tool, including the new modules for detecting advanced patterns, in order to determine whether the annotations (hints) in the code have been properly introduced.

4.4.1 Detection of the Pool pattern

To detect the Pool pattern, we leverage an application implementing the *traveling salesman problem* (TSP). Specifically, this NP-problem computes the shortest possible route among different cities, visiting them only once and returning to the origin city. This application leverages regular evolutionary algorithm that matches the Pool pattern, so it allows us to demonstrate how PPAT can detect these specific algorithmic instances.

Listing 4.1 shows the code snippet of the TSP problem matching a potential Pool pattern. As can be seen, the first stage of the main `while` loop contains the code related to the selection phase of the pattern. Concretely, for each population individual, the `for` loop computes its cost and picks the individual having the maximum cost. The second part of the `while` refers to the evolution function, where n swaps (mutations) on the selected individual are performed. The last code portion of loop takes the individuals in the populations and filters those with the maximum cost. This procedure is repeated until completing 2,000 iterations.

As observed in line 1, the new PPAT module for this pattern has properly identified and annotated a Pool and its corresponding functions, selection, evolution and filtering, in lines 3, 17 and 28, respectively.

4.4.2 Detection of the Image Convolution pattern

To test the detection of the Image Convolution pattern in PPAT, we use a synthetic benchmark computing the blur operation on an input image. In image processing, the application of a Gaussian blur filter (also known as Gaussian smoothing)

Listing 4.1: PPAT annotations for a code snippet matching the Pool pattern.

```
1  [[rph::pool, rph::id(0) ]]
2  while(2000 < count++){
3    [[rph::stage(0), rph::poolid(0), rph::in(population,problem),
4      rph::out(individual,population)]]
5    {
6      auto individual = population[0];
7      auto cost= std::numeric_limits<int>::max();
8      for( auto i = 0; i< population.size(); i++) {
9        auto aux = computeCost(population[i], problem);
10       if(aux < cost) {
11          cost = aux;
12          individual = population[i];
13       }
14     }
15     std::vector<std::vector<int>> selected;
16   }
17   [[rph::stage(1), rph::poolid(0), rph::farm, rph::in(individual),
18     rph::out(individual)]]
19   {
20     selected.push_back(individual);
21     for(int i = 0; i<num_swaps;i++){
22       int swap = i % (individual.size()-2);
23       auto aux = individual[swap];
24       individual[swap] = individual[swap+1];
25       individual[swap+1] = aux;
26     }
27   }
28   [[rph::stage(2), rph::poolid(0),
29     rph::in(individual,population,selected)]]
30   {
31     selected.push_back(individual);
32     auto cost = std::numeric_limits<float>::max();
33     auto best = 0;
34     for(auto i = 0; i<selected.size();i++){
35       auto aux = compute_cost(selected[i]);
36       if(aux < cost) {
37         cost = aux;
38         best = i;
39       }
40     }
41     population.push_back(selected[i]);
42   }
43 }
```

results into a blurred image by a Gaussian function. This filter corresponds with a convolution pattern, where each pixel in the output image is obtained using a neighborhood and the input kernel.

Listing 4.2 shows a code snippet corresponding to the computation of the Blur filter in our benchmark. As seen, this code is composed of four loops: the outer two iterating over the image pixels and the inner over the Blur kernel. This procedure computes the pixel values in the resulting image by applying such convolution operation. In this case, PPAT detects two convolutions placed in line 1 and 4, respectively. The first annotation identifies the convolution operation over the two

Listing 4.2: PPAT annotations for a code snippet matching the Image Convolution pattern.

```
1  [[rph::stencil, rph::convolution, rph::in(cols,image,frame,K),
2    rph::out(image)]]
3  for(int i = 1; i< rows-1; i++){
4    [[rph::stencil, rph::convolution, rph::in(image,i,frame,K),
5      rph::out(image)]]
6    for(int j= 1; j < cols-1; j++){
7      for(int ki = -1; ki < 1; ki++){
8        for(int kj = -1; kj < 1; kj++){
9          image[i][j]+= frame[i+ki][j+kj] * K[ki+1][kj+1];
10       }
11     }
12   }
13 }
```

dimensions in the image. The second annotation is detected only in one of the image dimensions. This occurs because PPAT is not able to detect how many dimensions the input data has. For instance the image could have been flattened and represented in a single vector instead of using a two-dimensional vector.

4.4.3 Detection of the Window Farm pattern

To test the detection of the Window Farm pattern in PPAT, we use a synthetic benchmark that computes average window values of the readings from an emulated sensor. The aim of the Window Farm pattern in this case is to reduce the frequency in which the items are delivered to the output stream.

Listing 4.3 shows the main business logic of this benchmark. As observed, the outer loop has been annotated by PPAT as a potential Window Farm. Inside this loop, there are three regions that correspond with the different Window Farm phases: window generation, kernel function and window update. In this concrete case, the window generation fills a buffer (window) with the items read from the emulated sensor (`read()` function call). The next region computes the kernel function, i.e. the average of a given window. The last portion code is in charge of updating the window by removing some of its items. As can be seen, these three stages have properly been identified by PPAT. Thanks to these annotations, a refactoring tool would be able to rewrite this code using, e.g., a parallel Window Farm pattern.

Note that the kernel function has also been annotated as a Farm pattern, as it can be computed in parallel by multiple threads with no side effects.

Listing 4.3: PPAT annotations for a code snippet matching the Window Farm pattern.

```
std::vector<int> window;
[[rph::window_farm, rph::id(0)]]
for(;;) {
  [[rph::stage(0), rph::wfarmid(0), rph::in(window), rph::out(window)]]
  {
    for(auto i = 0; i < 10; i++){
      window.push_back(read());
    }
  }
  [[rph::stage(1), rph::wfarmid(0), rph::farm, rph::in(window),
    rph::out(buffer,avg)]]
  {
    auto buffer= window;
    int avg= 0;
    for(auto i = 0 ; i < 10; i++){
      avg+= buffer[i];
    }
    avg= avg / buffer.size();
  }
  [[rph::stage(2), rph::wfarmid(0), rph::in(window,avg)]]
  {
    for(auto i = 0; i < 3; i++){
      window.pop();
    }
    out.write(avg);
  }
}
```

5. Software availability

The PPAT software implementation is available at the UC3M git repository:

https://github.com/arcosuc3m/ppat

This software is released under an open source license.

6. Conclusions and future works

In this deliverable, we have reviewed the state-of-the-art about and existing program shaping and pattern detection tools. Next, we described the set of program shaping advanced techniques and methods in order to refactor sequential C++ programs into hygienic C++ code with equivalent functionality. The deliverable also describes the conditions and requirements that need to be satisfied for the **RePhrase** advanced pattern set. These conditions allow to determine when a portion of sequential code may match a given advanced patterns. Finally, we extend our Parallel Pattern Analyzer Tool with the modules for detecing the Pool, Image Convolution and Window Farm advanced parallel patterns.

The contributions in this deliverable involve the Parallel Pattern Analyzer Tool (PPAT) with modules for detecting some of the advanced patterns: Pool, Image Convolution and Window Farm. These modules allow the tool to annotate code regions (loops) when the requirements established are met. As noted in the previous chapter, the inherent complexity of these advanced patterns prevents PPAT from guaranteeing the correctness of the detections. This mainly occurs because these patterns can be implemented in many different ways, code arrangements, so by nature, are complex to identify in sequential codes. Therefore, annotations introduced by the tool in these cases should be taken as mere hints.

As future work we plan to improve the detection mechanisms implemented in these modules so as to guarantee the correctness of the detections. Furthermore, we will add support to decide which is the most suitable pattern to be introduced when more than one pattern can be a candidate for a given portion of code.

Bibliography

[1] PoCC: the polyhedral compiler collection version 1.2. http://www.cse.ohio-state.edu/~pouchet/software/pocc/, 2013. [Last access 7th January 2016].

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, pages 261–280. John Wiley & Sons, Inc., 2017.

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

[4] István Bozó, Viktoria Fordós, Zoltán Horvath, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. Discovering parallel pattern candidates in erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 13–23, New York, NY, USA, 2014. ACM.

[5] Doina Bucur, Giovanni Iacca, Giovanni Squillero, and Alberto Tonda. *An Evolutionary Framework for Routing Protocol Analysis in Wireless Sensor Networks*, pages 1–11. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[6] Murray I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Research monographs in parallel and distributed computing. Pitman Cambridge, MA, London, 1989.

[7] Johan Enmyren and Christoph W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[8] Ewa Gajda-Zagórska. *Multiobjective Evolutionary Strategy for Finding Neighbourhoods of Pareto-optimal Solutions*, pages 112–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[9] ISO/IEC. Information technology – Programming languages – C++. International Standard ISO/IEC 14882:20111, ISO/IEC, Geneva, Switzerland, August 2011.

[10] Xiaoming Li, Jack B. Dennis, Guang R. Gao, Willie Lim, Haitao Wei, Chao Yang, and Robert Pavel. FreshBreeze: A Data Flow Approach for Meeting DDDAS Challenges. *Procedia Computer Science*, 51(Complete):2573–2582, 2015.

[11] Zhen Li, Rohit Atre, Zia Ul-Huda, Ali Jannesari, and Felix Wolf. DiscoPoP: A profiling tool to identify parallelization opportunities. In *Tools for High Performance Computing 2014*, chapter 3, pages 37–54. Springer International Publishing, August 2015.

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

[13] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[14] W. M. Miller. A Taxonomy of Expression Value Categories.10-0045=N3055. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3055.pdf Accessed 6th May 2015, 2010.

[15] Korbinian Molitorisz, Tobias Müller, and Walter F. Tichy. Patty: A pattern-based parallelization tool for the multicore age. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 153–163, New York, NY, USA, 2015. ACM.

[16] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[17] Vladan Popovic, Kerem Seyid, Eliéva Pignat, Ömer Çogal, and Yusuf Leblebici. Multi-camera platform for panoramic real-time hdr video construction and rendering. *Journal of Real-Time Image Processing*, 12(4):697–708, 2016.

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

[19] *EU Project "REPARA: Reengineering and Enabling Performance And poweR of Applications"*, 2015. http://repara-project.eu/.

[20] *EU Project "RePhraseRefactoring Parallel Heterogeneous Resource-Aware Applications - a Software Engineering Approach"*, 2016. http://rephrase.weebly.com/.

[21] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Computing*, 36(9):531 – 551, 2010.

[22] L. M. Sanchez, J. Fernandez, R. Sotomayor, S. Escolar, and J. D. Garcia. A comparative study and evaluation of parallel programming models for shared-memory parallel architectures. *New Generation Computing*, 31(3):139–161, 2013.

[23] Georgios Tournavitis and Björn Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 377–388, New York, NY, USA, 2010. ACM.