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 initial patterns
D2.3

Due date of deliverable: M12

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

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

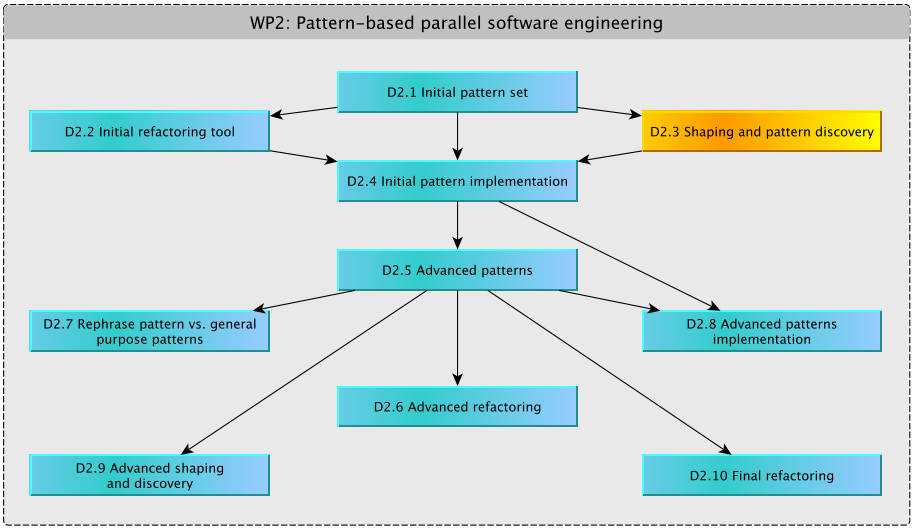
Rev.	Date	Who	Site	What
1	15/2/16	Manuel F. Dolz, J. Daniel Garcia	UC3M	Submitted first version/PPAT tool
2	18/2/16	Evgueni Kolossov	PRL	Added chapter for programming methods and techniques
3	20/2/16	Zoltan Theisz	Evopro	Included evopro use case for PPAT evaluation
4	10/10/16	Marco Danelutto	UNIPI	Fixed typos and formatting
5	20/02/17	Marco Danelutto	UNIPI	Added placement picture in executive summary

Executive Summary

This document is the third 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.

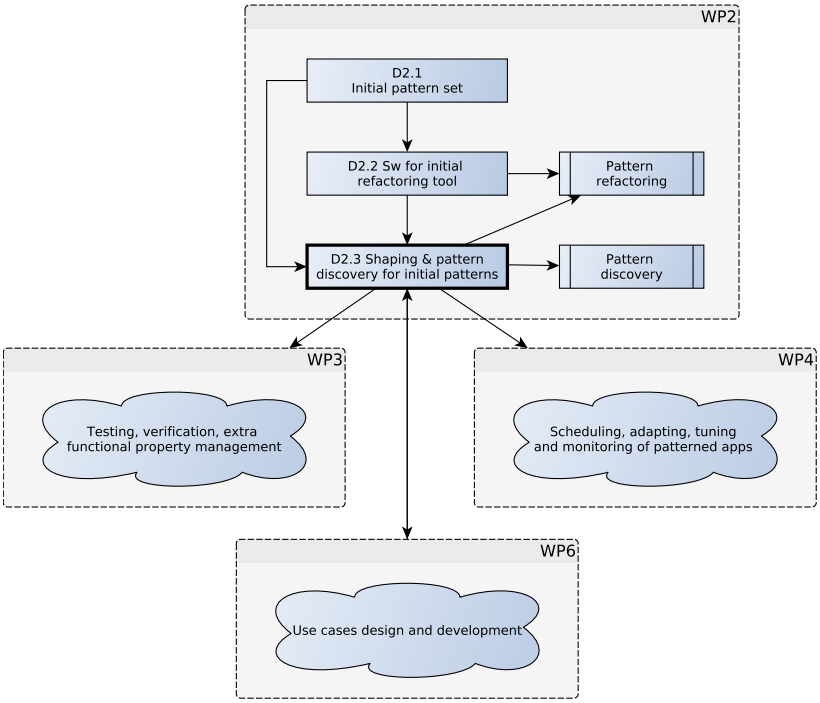
This deliverable, D2.3 “Report on program shaping and pattern discovery for the initial pattern set.", integrates the results of the different phases of WP2 (T2.3 “Program Shaping” and T2.4 “Pattern Discovery”) where, according to the DoW *we will report the initial program shaping methods* and *describe the pattern discovery techniques* for the initial pattern set. The work in this deliverable is divided in two steps. First, it defines the program shaping techniques and methods for the initial 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 presents a prototype for pattern discovery that, using the aforementioned conditions, identifies instances of parallel pattern candidates within C++ applications.

The placement of D2.3 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 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 initial 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 pattern candidates within C++ applications at compile time.
- Evopro has contributed in Chapter 4, evaluating the parallel pattern discovery tool on the main code base of evopro eRDM use-case.
- UC3M has produced Chapter 5 enumerates a few concluding remarks and future works.

Contents

1. Introduction

Although most of the current computing hardware, such as multi-/many-core processors, GPUs or co-processors, has been envisioned for parallel computing, much of the prevailing production software is still sequential [26]. In other words, a large portion of the computing resources, i.e. cores, provided by modern architectures are basically underused. Therefore, a big effort in this sense needs to be carried out by the HPC community so as to refactor sequential software into parallel. To tackle this issue, several solutions in the area have been implemented. For instance, parallel programming frameworks and concurrent skeletons are well-known approaches in order to efficiently exploit parallel computing architectures [7, 23]. Actually, in the current state-of-the-art there can be found multiple parallel programming frameworks that benefit from shared memory multi-core architectures, such as OpenMP [21], Cilk [6], Intel TBB [27] or FastFlow [9]; distributed platforms, such as MPI or Hadoop; and some others especially tailored for accelerators, as e.g., OpenCL and CUDA. Nevertheless, only a small portion of production software is using these frameworks.

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. Practical use cases in this sense are sequential data-intensive applications, broadly encountered in production scientific and industrial areas. Just a simple analysis over their code would reveal that a vast majority of algorithms and methods used underneath match perfectly with parallel patterns. A solution to parallelize these codes is to manually translate the source file codes, however this task results, in most cases, a cumbersome and very complex task for large applications. Another solution is to use refactoring tools, applications that advice developers or even semi-automatically transform sequential code into parallel [5, 8]. Although source codes transformed using these techniques do not often get the best performance, they aid to a great extent developers in order to reduce necessary refactoring time [17]. Unfortunately, refactoring tools found are still premature, not yet being fully adopted by development centers. In fact, many of them are human-supervised, being the developer responsible for providing specific sections of the code being analyzed and refactored. Although this alleviates the source-to-source transformation, the process remains semi-automatic. A key component for turning this process from semi- to full-automatic are parallel pat-

tern detection/discovery tools.

This document defines the set of 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. It also describes the sets of requirements and properties for the initial pattern set, defined in D2.1, that need to be satisfied in C++ codes in order for a particular pattern to be introduced. The rules stated for these patterns will be further used to analyze patterns from the use cases provided by developers in the project and to sketch their parallel implementation. Also, they will be leveraged in the "core" software engineering related activities of the project to focus generic software engineering techniques on *structured* parallel software rather than on generic parallel software. Furthermore, we provide a prototype module extension for detecting the pipeline pattern along with an evaluation with a set of state-of-the-art sequential benchmarks and of some selected tests on code-bases from industrial solutions. In general, all these facts motivate the contributions presented in this deliverable, namely D2.3, 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 initial set of **RePhrase** the conditions and requirements that need to be satisfied in order to be introduced in C++ applications.
- Chapter 4 presents a prototype pattern discovery tool that, using the aforementioned conditions and a static approach, identifies instances of pattern candidates within C++ applications at compile time.
- Chapter 5 enumerates a few concluding remarks and future works.

2. Program shaping methods and techniques

2.1 Introduction

This part of the document identifies the unhygienic code properties, such as side-effects and unnecessary task/data dependencies. This approach is independent of the refactoring method used, allowing for such refactoring to be automatic or manual. Rectifying unhygienic code properties increases the proportion of source code to which parallel patterns can be applied.

The unhygienic properties are categorised in the following sections.

2.2 Identification of poor coding practice and undefined behaviour

Undefined behaviour affects code in several ways by:

1. invalidating or corrupting the object model.
2. introducing non-deterministic behaviour.

In a sequential program a minor misuse of the memory model may well go unnoticed; however, it is much more likely to impact a parallel program. Such issues must be addressed. Examples of such behaviour are:

- Use of an object after its lifetime has ended.
- Use of an invalid memory (e.g. dereference of null pointer, or outside of an array bounds)
- Invalid arithmetic operation (e.g. mod/divide by zero)
- Missing *return* statement.
- Use of an object with *indeterminate* value.

The second category of undefined behaviour may not be fatal but can result in non-deterministic program behaviour. Examples are:

- Dependence on the sequence of evaluation within an expression.
- Use of relational or binary minus operator on pointers to different objects.

In addition to the above, use of certain best practices simplify the identification of *hygienic* code:

- Use const container calls when result is immediately converted to a const iterator (e.g. cbegin/cend)
- Declare static a member functions that does not refer to *this*.
- Declare const a member functions that refers to *this* but does not modify the objects externally visible state.
- Postpone variable definitions as long as possible.
- Use RAII for resources.
- Ensure that all statements are reachable and no expression or sub-expression is redundant.

2.3 Identification of pure functions

This section outlines properties a function must hold for it to be *pure*, thereby meeting a precondition for many parallel patterns.

2.3.1 Non-pure reads and writes

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.

The following is an example of a pure function:

```
int * f2(int i, int j)
{
  int * k = new int (i + j);
  return k;
}
```

For determining if a function is pure or not, taking the address of a variable will be considered as a read of the variable itself. For example, reading the address of a variable with static storage duration will result in the function being non-pure:

```
int m;
int * f2()  // non-pure as 'reads' m
{
  return &m;
}
```

2.3.2 Non returning functions

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`

When determining if a function returns, exceptions are not considered.

2.3.3 Calls made to non-pure functions

A function will inherit the status of functions it calls. Therefore, a call to a non-pure function will result in the caller also being non-pure.

```
void f1();  // non-pure

void f2()   // not pure, calls 'f1()'
{
  f1();
}
```

2.3.4 Code reachability

For the purpose of determining if a function is pure, the reachability of non-pure code will be ignored. For example:

```
void f1();  // not pure

void f2()   // not pure, calls 'f1()'
{
```

```
  if (0) {
    f1();
  }
}

void f3()   // not pure, calls 'f1()'
{
  return ;
  f1();
}
```

However, note that unreachable code is in itself unhygienic, see Section 2.2

2.4 Identification of Pure Functors

This section outlines properties of a functor that would result in it not being a pure functor, thereby invalidating a precondition for many parallel algorithms.

A functor or function object is a class that implements the *function call operator*. Additionally, the function object can also contain state.

2.4.1 Context capture

A functor can be implemented in several ways:

- a class defining the *function call operator*, or
- a *lambda expression*, or
- binding arguments to a function using `std::bind`

In each of these cases, context from the enclosing scope can be captured in the functor, either by reference or by value/copy. Any capture by reference will result in the functor being non-pure.

2.4.2 Non-pure reads and writes

Similar to the restrictions for *pure functions*, a functor will not be pure if its *function call operator* reads or writes objects other than:

- a non-volatile automatic variable, or
- a function parameter, or
- an object allocated within the body of the function, or
- the captured context (read only).

2.4.3 Calls made to non-pure functions

A functor will inherit the status of functions it calls in its *function call operator*. Therefore, a call to a non-pure function will result in the caller also being non-pure.

2.4.4 Exceptions

If a functor is used in an algorithm, throwing an exception from a functor will result in the algorithm to be aborted. Therefore, the use of exceptions will result in the functor being non-pure.

2.4.5 Non returning functors

Restrictions of Section 2.3.2 apply to functors also.

2.5 Identification of unhygienic loops

The smallest section of source code which can be parallelised is an innermost loop. However, a loop can exhibit unstructured properties, which may make it impossible to meaningfully speed up, or achieve equivalence with the sequential version.

If a loop initialisation statement, condition or increment expression contain side-effects, these will occur before evaluating the loop body on each iteration, and are considered when determining if a loop is hygienic.

2.5.1 Accessing objects visible outside of the loop

A data race or loop carried dependence is possible, when parallelising a loop body that accesses any object with lifetime exceeding that of the loop itself. Both stack and heap allocated objects are considered. For the purposes of determining lifetime, usage of the *address of* operator, memory allocation function, or *copy assignment* are considered the only valid ways to set the value of a pointer, e.g. the effects of pointer arithmetic are ignored. Both the lifetime of a pointer and its aliased object are taken into account.

2.5.2 Jumps

Presence of *break*, *continue* or *return* statements in a loop violates a precondition for many parallel patterns. Additionally, a *goto* statement jumping outside of the loop will similarly make the loop unhygienic; as will a non-returning construct appearing in the body of a loop, as defined in Section 2.3.2. Note, that an infinite loop is hygienic, as it can be parallelised, as opposed to a loop with one of its iterations never terminating.

A backward goto, i.e. jumping to a preceding location in source code, not crossing a loop scope is equivalent to a loop itself, and is considered unhygienic. The loop should be rewritten with an equivalent explicit loop, if a parallel pattern is to be applied to it.

2.5.3 Throw

Exceptions are assumed to rarely occur, so typically they can be ignored, or handled at the top level of each thread. However, exceptions could also be misused for regular control flow, e.g. in place of a forbidden *break* or *goto* statement. With this in mind, occurrence of a *throw* statement in a loop is also unhygienic.

2.5.4 Calling pure functions

Another prerequisite of many parallel patterns is that all functions called from a loop are pure.

2.5.5 Nested loops

An outer loop may be hygienic even though an inner loop is not, for example:

```
void pure1(int a);

void f1(int a)
{
  for (int i (0); i < a; ++i)
  {
    int n (0);

    for(int j (0); j < i; ++j)
    {
      if(5 == j)
      {
        continue; // inner loop is unhygienic
      }
      ++n;        // inner loop is unhygienic
    }

    pure1(n);    // outer loop is hygienic
  }
}
```

Providing the outer loop catches all exceptions thrown explicitly by any inner loops, it will remain hygienic. Note, that this concept does not extend to calls to non-pure functions, or a loop with an inner infinite loop, which propagate the unhygienic classification to all enclosing loops and functions.

2.6 Identification of unhygienic algorithm use

Some uses of sequential STL algorithms can be trivially parallelised if the functors used don't exhibit unstructured properties. The use of non-pure functors in sequential algorithms is therefore unhygienic.

2.7 Identification of the function cost

An ability to estimate the runtime cost of a loop or a function can augment or replace the need to perform profiling, to identify the best parallelisation opportunities. The techniques for compile time cost estimation are detailed below. Note that it is sufficient to estimate the relative cost in terms of a unit, e.g. the simplest possible CPU instruction modifying the program state, such as a memory fetch, ignoring the effects of compiler optimization.

2.7.1 Estimating the cost of statements

Expression and declaration statements are the building blocks of a C++ program, with all other kind of statements grouping these into well defined control flow. Additionally, the body of a function is a compound statement, and a function call is an expression. Starting from a particular entry point, a cost estimate can be computed by recursively applying formula for accumulating the cost of constituent statements and expressions, as detailed below.

Expression cost: The basis for estimating cost of a particular expression can be derived from *ISO/IEC 14882:2011 Programming Language C++*, relative cost of individual instructions on a von Neumann architecture, and for higher level C++ constructs from *ISO/IEC TR 18015: Technical Report on C++ Performance*. With the exception that the cost of a function call expression is the cost of the associated function definition – its compound statement. Any recursive application of function cost is prevented, and lower bound of the cost of a single recursive pass is used.

The cost of an expression statement is simply the cost of the associated expression, as is the cost of a declaration statement with an initialiser, e.g. the cost of a constructor and destructor call. A declaration of a *POD* object without storage duration has no cost.

Compound and label statements: The cost of a compound statement is a sum of the cost of its constituent statements. Similarly, the cost of a label statement is that of its sub-statement.

Selection statements: The cost of an *if* statement is taken to be the maximum cost of any of its sub-statements, added to the cost of the condition. Similarly, for a *switch* statement, with the sub-statements taken to be the individual case clauses, accounting for the effect of any fall-through.

Iteration statements: If the number of loop iterations can be determined at compile time, the cost of a loop is defined by the following formula $a + n * (b + c + d)$, where a is the cost of an initialisation statement, if any, b is that of a condition expression, c – increment expression, if any, d – the loop body, and n is the number of iterations. The cost of other loops is calculated with the above formula assuming

a single iteration ($n = 1$), to provide a lower bound for deciding to apply parallel patterns.

Jump statements: The cost of a *goto* statement is a single unit of cost. Although, a backward goto is equivalent to a loop, this estimate provides a lower bound, taking into account that the loop is unhygienic (see Section 2.5.2), so only the enclosing loop or function may be a candidate for parallelisation.

2.7.2 Identification of initialisation code

Typically a program will consist of some initialisation code, followed by a form of significant iterative computation, and finally clean-up code. The initialisation and clean-up code are not particularly promising areas for parallelisation, thus if suitably identified, that code can be excluded from application of parallel patterns.

Initialisation code is identified as code from the beginning of the main function, up to the first occurrence of a nested loop or a loop with an unknown number of iterations (see Section 2.7.1). Similarly, the clean-up code extends from the last such occurrence (excluding it) to the end of the main function. For the purposes of this analysis, function calls are replaced with the corresponding function body.

2.8 Decidability

Identification of unhygienic code properties specified above can be performed with suitable static code analyses. In particular *pattern based* analysis applied to a complete program is sufficient to accurately determine all unhygienic code, and calculate function cost. Therefore, using the terminology of MISRA-C guidelines, all Program Shaping methods defined in this report are *decidable*.

3. Pattern discovery techniques

This part of the document enumerates the properties and conditions for each of the patterns of initial set from the **RePhrase** project, that need to be satisfied in order to be applied into a C++ application. This 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 D2.1 for describing their requirements. To ease the understanding, we reference each pattern described with references to D2.1. We refer these parallel design patterns "fundamental" as we aim at including in the initial set the most common and throughly used known parallel design patterns. Note that the most important patterns are Farm, Pipeline, Map, Stencil and Reduce, and others, such as Stream filter pattern can be derived from the previous ones.

The initial set of **RePhrase** patterns is described both classifying the patterns according to the kind of parallelism captured (stream or data parallelism) and distinguishing the way data to be processed are provided to similarly structured patterns (from external or internal stream sources or from existing data collections, either in-memory or disk based). We also list "sequential" patterns with the purpose of providing the application programmer with patterns suitable to wrap existing sequential ("business logic") code in such a way the code may be used as functional parameter of other patterns (e.g. a stage in pipeline or a worker in a map pattern). In the descrition of the patterns we also relate data management patterns that can be introduced during the refactoring task.

3.1 Sequential patterns

3.1.1 Sequential code wrapper

This pattern wraps a sequential code portion such that it may be used to compute the application "business logic" in all those places where parallel pattern require a parameter expressing a computation. More details about this pattern can be found in Deliverable 1.2, Section 3.1.1.

Requirements The requirements of this pattern should determine whether a portion of code can be translated into a pure function. The requirement of this pattern

is summarized as follows:

- *No side effects*. There should no exist read (write) accesses to variables that are not included in the input (output) paramters or declared internally, nor static variables representing internal states.

3.1.2 Sequential composition

This pattern executes two or more computations using the same resources, sequentially. More details about this pattern can be found in Deliverable 1.2, Section 3.1.2.

Requirements The requirements of this pattern should determine whether a portion of code can be translated into a sequence of pure functions. The requirements of this pattern are summarized as follows:

- *No side effects*. There should no exist read (write) accesses to variables that are not included in the input (output) paramters or declared internally, nor static variables representing internal states.

- *Set of pure functions*. The portion of the code should be divisible in more than one pure functions.

3.2 Stream parallel patterns

In this section we describe the requirements of the *stream parallel* patterns included in the initial **RePhrase** pattern set. Basically these patterns exploit parallelism in the processing of different items belonging to one or more input data streams. An input data stream is characterized by having a type and by being able to provide items (to be computed) one after the other with a given *interarrival time*.

The stream parallel patterns included in the initial **RePhrase** pattern set include patterns modelling *map*, *filter*, *reduce* and *iterative* computations.

3.2.1 Farm

This pattern computes in parallel the same function $f : \alpha \rightarrow \beta$ over all the items appearing onto an input stream of type α stream delivering the results on the pattern output stream of type β stream. Computations relative to different stream items are completely independent. This pattern is also referred to as *task farm* or *stream map*. During the refactoring process, the introduction of this pattern implies the use of a stream generator and collapser patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.2.1.

Requirements The requirements of this pattern should determine whether a loop can be refactored into a parallel farm, i.e., if its iterations are independent each other. These requirements are described as follows:

- *No global variables modified.* There should not exist instructions that modify global variables in the loop.
- *No RAW dependencies*. There should not exist Read-After-Write (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.

3.2.2 Pipeline

This pattern computes in parallel several stages on a stream item. Each stage processes data produced by the previous stage in the pipe and delivers results to the next stage in the pipe. During the refactoring process, the introduction of this pattern implies the use of a stream generator and collapser patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.2.2.

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

- *Multiple stages*. The loop should be able to be divided in more than one stage.
- *Interconnected stages*. Each stage in the pipeline should receive, as inputs, the outputs from the previous stage.
- *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 simultaneously within stages of the pipeline.
- *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.

3.2.3 Stream filter pattern

This pattern computes in parallel a filter over an input stream of type α stream, that is passes to the output stream of type α stream only those input data items passed by a given boolean "filter" function (predicate) $\mathcal{P} : \alpha \rightarrow \{true, false\}$.

During the refactoring process, the introduction of this pattern implies the use of a stream generator and collapser patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.2.3.

Requirements The requirements of this pattern can be summarized as follows:

- *Boolean filter function*. The filter function of the stream filter should be a pure boolean function, i.e., the result of the function on a stream element should be independent from other stream elements. This function should receive only one element of the input stream.
- *Equal input/output stream types*. The input and output stream data type should be the same.
- *Conditional output generation*. There should exist a conditional statement that determines which input elements are passed to the output stream. This statement depends on the boolean filtering function.

3.2.4 Stream accumulator pattern

This pattern "sums up" all items appearing on the input stream and delivers results to the output stream. The function used to sum up values ($\oplus$) may be any kind of binary function of type $\oplus : \alpha \times \alpha \rightarrow \alpha$. During the refactoring process, the introduction of this pattern implies the use of a stream generator and collapser patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.2.4.

Requirements The requirements of this pattern can be summarized as follows:

- *Accumulative function*. The accumulative function should be a pure function computing a binary operation that, applied to a subset of the input stream elements, produces an output element.
- *Equal input/output stream types*. The input and output stream data type should be the same.

3.2.5 Stream iteration pattern

This pattern implements a function α stream $\rightarrow \alpha$ stream by iterating the computation of another pattern α stream $\rightarrow \alpha$ stream over one or more items appearing onto the input stream, and delivers results on the output stream. During the refactoring process, the introduction of this pattern implies the use of a stream generator and collapser patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.2.5.

Requirements The requirements of this pattern 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.
- *Boolean expression or 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.

3.3 Data parallel patterns

In this section we describe the requirements of the *data parallel* patterns included in the initial pattern set. Basically these patterns exploit parallelism in the processing of different items or (possibly overlapping) partitions of items belonging to a single "collection" data item. The different data processed in parallel exist at a given point in time, that is there is no need to await them in time as it happens for stream data items.

The data parallel patterns included in the initial pattern set include *map*, *reduce*, *stencil*, *divide and conquer* and *iterative* computations.

3.3.1 Map

This pattern computes a given function $f : \alpha \rightarrow \beta$ over all the data items of an input collection whose elements have type α and produces as output a collection of items of type β hosting the resulting values isomorphic to the input collection. Each item at a generic position i in the output collection come from the computation of the function f onto the data item in the corresponding position of the input collection. This patterns is also known as *parallel for*, *apply-to-all*. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.3.1.

Requirements The requirements of this pattern should determine whether a loop can be refactored into a parallel map, i.e., if its iterations are independent each other and if there are relationships between the input and output data streams. These requirements are described as follows:

- *Dataflow dependencies*: Each element of the output stream should depend on, at least, one element of the input stream.
- *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 in different 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.

3.3.2 Stencil

This pattern computes in parallel the new value of items in an input data collection to be placed at the correspondent position into an isomorph output collection. The computation of the result relative to the item requires as input data some items belonging to the nearer positions of the input collection. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.3.2.

Requirements The requirements of this pattern should determine whether a loop can be refactored into a parallel stencil, i.e., if its iterations are independent each other and if there are neighborhood relationships between the input and output data streams. These requirements are described as follows:

- *Neighborhood dependencies*: Each element of the output stream should depend on, at least, one element of the input stream and its neighbours.
- *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.

3.3.3 Reduce

This pattern "sums up" all the data items of a collection of items of type α using a binary function $\oplus : \alpha \times \alpha \rightarrow \alpha$ that is usually associative and commutative. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.3.3.

Requirements The requirements of this pattern can be summarized as follows:

- *Accumulative function*. The accumulative function should be a pure function computing a binary operation that, applied to the set of the input stream elements, produces an output element.
- *Accumulative variable*. The accumulative variable is the output value and it is accessed for read and write each time the accumulative function is called.
- *Equal input/output stream types*. The data type of the input stream and output value should be the same.

3.3.4 MapReduce

This pattern computes a key value function over all the items of an input connection and eventually delivers a set of unique key value pairs where the value associated to the key is the "sum" of the values output for the same key in the first "map" phase. This pattern is also known as Google mapreduce. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.3.4.

Requirements The detection of the MapReduce pattern is basically a composition of stream filters and a reduce pattern, therefore their individual requirements should be checked. These requirements are described as follows:

- *Three stages*. This pattern should be composed of three stages: two map stages and a reduction stage.
- *Key production*. The first stage should process each input data item and produce a $\langle$*key, value*$\rangle$ pair out of it. This stage should meet the requirements relative to farm workers, that is:
  - *No global variables modified*. The code producing the $\langle$*key, value*$\rangle$ pairs out of the input data should not modify any global variable.
- *Key agglomeration*. The second stage should process all the $\langle$*key, value*$\rangle$ pairs to produce a collection of $\langle$*keyX, [valueX*$_1$*, ... , valueX*$_{n_x}$*]* $\rangle$ records, each hosting all the values appearing in the $\langle$*key, value*$\rangle$ pairs in input relative to the same key *keyX*. The stage should meet the requirements of a stream accumulator stage[1].

[1] actually, all but the requirement relative to input/output types: in the stream accumulator stage these types should match, in this case the input type will be $T_{key} \times T_{value}$ while the output type will be instead $T_{key} \times (T_{value})^\star$

- *Key reduction*. The third stage should process each item $\langle$*keyX, [valueX*$_1$*, ... , valueX*$_{n_x}$*]*$\rangle$ summing up the values to produce an output item $\langle$*keyX, sumValueX*$\rangle$. This stage should meet the requirements relative to farm workers, that is:
  - *No global variables modified*. The code producing the $\langle$*key, value*$\rangle$ pairs out of the input data should not modify any global variable.
- *Output tuple*. The output of this pattern should be a set of unique key value tuples where the keys are the ones produced during the first stage and the values are those computed (for each key) during the reduction stage.

It is worth pointing out that a mapreduce pattern may be discovered also from code in two stages (only) where the first is the very same first stage of the three stage decomposition discussed above (and therefore has the same requirements), and the second one is a stage computing the accumulation of values for the same key while gathering them from the input pairs, being *de facto* the union of the second and third stages of the three stage decomposition just discussed. In this case this second stage will need all the requirements of a stream accumulator pattern are meet.

3.3.5 Divide and conquer

This pattern computes a problem for which *a)* the solution for some base cases are known and *b)* non-base case problems may be divided into a collection of sub-problems and the solution of the non-base case problems may be computed out of the solutions of the sub-problems. During the refactoring process, the introduction of this pattern implies the use of a splitter and merger patterns. More details about this pattern can be found in Deliverable 1.2, Section 3.3.5.

Requirements The requirements of the divide an conquer pattern should determine whether there exists a portion of the code calling a recursive function whose body is logically divided in stages:

- *Termination*. A boolean parameter is evaluated on the input parameter. If true, a function of the input parameter is returned as function result.
- *Divide phase*. The input parameter is processed to produce a set of parameters of the same type.
- *Recursive call*. A loop iterates over all the parameters in the set recursively calling the procedure with one item in the set as input parameter.
- *Conquer phase*. A loop iterates over the results of the recursive calls to "accumulate" (conquer) the function result.

The requirements of the different phasses may be described as follows:

- *All phases. No global variables are modified.*
- *Recursive call* and *conquer phase*. These are loops with *no break statement* in the loop body.
- *Divide phase*. The input type is T_{input} and the output type is $(T_{input})^\star$.

3.4 Notes on code requirements for pattern discovery

The requirements on the source code listed in the sections above are relative to the discovery of patterns whose business logic code is actually *sequential* code. In the most general case, parameters of the patterns may also be other patterns. As an example, a pipeline stage may be implemented as a map (or farm) pattern to further improve parallelism explotiation and, consequently, performances in all those cases where the pipeline stage results to be a bottleneck.

As such, the requirements listed above refer to properties that *must* be ensured on the sequential code such as the code begin side effect free or with no control flow break statements. A further requirement may be added to most pattern requirement sections stating that each one of the pattern functional components may in turn be a pattern and therefore should recursively fulfill the corresponding requirements. The pipeline stage modelled after a farm should therefore ensure the requirements for the farm workers are fulfilled and this automatically qualifies the farm pattern as a pipeline stage fulfilling the pipeline stage requirements.

In some cases, the identification of loops as potential places where patterns may be discovered is to be intended as a particular case of a more general situation where separate portions of code produce "streams" or "data collections" that are subsequently processed by stream or data parallel patterns. As an example, in a code structured such as follows:

```
TypeA x[N];
TypeB y[N];

for(int i=0; i<N; i++) {
  // fill some vector
  x[i] = ...;
}
for(int i=0; i<N; i++) {
  // process vector items (independent iterations)
  y[i] = f(x[i]);
}
```

we main aim at discovering two distinct pattern structures:

- a stream generator pattern, followed by a farm pattern computing function `f`, or

- a single pipeline pattern with two stages corresponding to the two for loops in the code.

Whatever the pattern discovery results is, the refactoring rules will eventually be able to reconduct the first "patternization scheda" to the second one and viceversa.

Last but not least, it is worth pointing out that there are particular properties that must be ensured in a number of different patterns, such as the absence of `break` statements, the absence of statements modifying global variables, or the absence of RAW dependencies in loop iterations. Such properties clearly require special code to be tested quickly and effectively on those portions of code individuate as pattern "candidates".

4. Parallel pattern discovery tool

In this part of the document we propose the Parallel Pattern Analyzer Tool (PPAT) as a prototype tool for detecting 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 guarantees that parallel patterns detected comply with a series of requirements that prove the correctness of the solution.

4.1 Related work

In the current state-of-the-art, there can be found multiple works related to automatic detection and refactoring of sequential source codes based on parallel patterns. However, this task is not simple and can be treated using different approaches. At this time, tools developed to detect the pipeline parallel pattern on sequential codes are tied to the programming language used and most of them require profiling techniques in order to check associated data-flow dependencies.

In the literature, we find tools similar to ours based on static analyses of the code to detect parallel patterns, but that require profiling techniques during the identification step. For example, the approach developed by Sean Rul et al. [22] 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. This tool, however, presents some limitations: it needs to execute the target application several times to profile it, and it is tied to the C programming language. Contributions such as the work by Molitorisz et al. [19], detect statically potential pipelines, nevertheless they do not check for dependencies, so the correctness of the resulting parallel application cannot be guaranteed. Instead, it checks for data races and dependencies on a subsequent execution and profiling process. Contrary to that, 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, e.g., maps, stencils, pipelines etc.

On the other hand, we encounter tools that detect parallel patterns using only

profiling techniques. For example, DiscoPoP in [10, 14] leverages dependency graphs in order to detect parallel patterns. Nevertheless, the tool has some drawbacks: the profiling techniques have a non-negligible execution time and memory usage. A similar approach by Tournavitis et al. [28] detects and transforms legacy code into parallel using parallel patterns. Alternatively, FreshBreeze [13], a dataflow-based execution and programming model and computer architecture, leverages similar static loop detection techniques that analyze dependencies and transform parallelizable loops using the task model and tree-structured memory proposed by the system. It is important to remark that approaches based on static analysis are not very extended in the area, since analysis of data dependencies at compile-time becomes a much more complex task. Instead, some works leverage 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's 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. In order to transform legacy code using parallel patterns, different methodologies have been proposed. For example, Massingill et al. [15] present a reengineering process to aid the parallelization process of legacy code. In a similar way, Jahr et al. [12] propose a *pattern-supported parallelization approach*, i.e, a methodical model-based design approach to be executed by an engineer or software developer. This workflow allows introducing parallelism in an application only by defining building blocks [16].

The parallel pattern detector prototype presented in this report differenciates 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 [20]. 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 [18], 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.

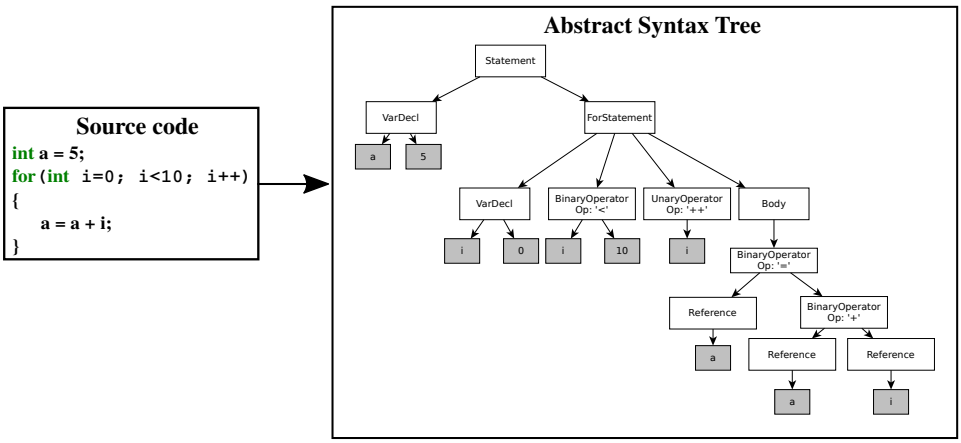


Figure 4.1: Example of Abstract Syntax Tree for a given source code.

- 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.

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 Analyzer Tool

In this section we introduce the Parallel Pattern Analyzer Tool (PPAT) in detail. 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. For brevity reasons, we only present the module for detecting the parallel pipeline pattern.

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 refactored into a parallel pattern. Basically, it iterates the AST, extracts loop-related subtrees and gathers information about possible variables, function calls, nested loops, conditional statements, etc. coded into the loops. On the other hand, it collects information about the functions implemented in the user code.

2. *Feature extraction.* This step leverages structures collected in the previous step in order to extract specific features about iterators, variable declarations,

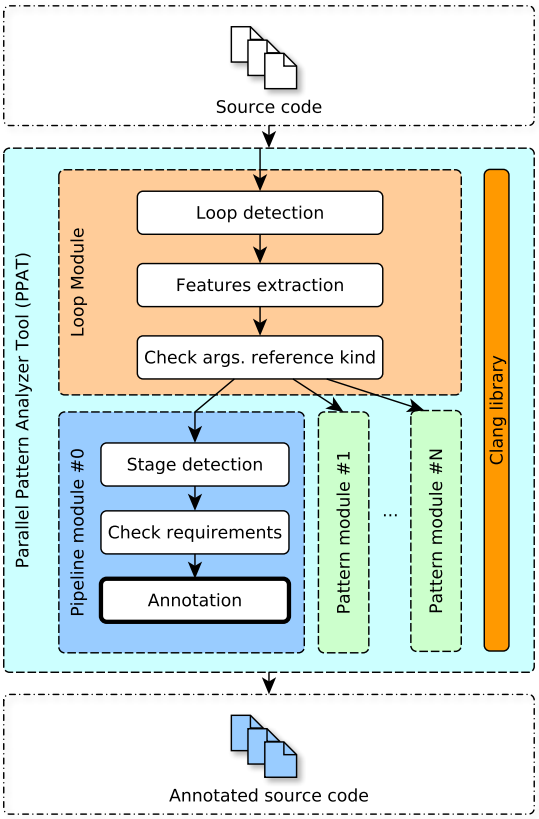


Figure 4.2: Workflow diagram of PPAT.

references, function calls, inner loops, memory accesses, operations, etc. Next, for each statement encountered, we store, using a set of C++11 structures, information about location on the original code, variable and functions name, variable kinds (Lvalue or Rvalue) 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 is Rvalue or Lvalue. When this occurs, the tool performs the following actions:

   (a) If the function code is available or the function is implemented on the user code, it is possible to check the set of arguments and assign the right variable kind (Lvalue or Rvalue). If the argument is not modified, it is considered as Rvalue. In contrast, if there exist write accesses to the variable, the Lvalue is assigned as the variable kind. Alternatively, if there is a RAW dependency on the variable, the kind is set to L/Rvalue, since the argument can generate a potential feedback be-

tween iterations.

(b) On the contrary, if the function cannot be accessed, it is not possible to check the actual variable kind. Thus, the tool takes a conservative decision: it set the arguments kinds always to L/Rvalue. Despite of this, it inserts the function name and arguments kinds into a dictionary file in order to improve the detection process in future analysis. Afterwards, the user can eventually manually modify this dictionary to set the right arguments kinds of these specific functions.

Finally, marked loops are passed to the different pattern analyzer modules responsible for detecting parallel patterns of the **RePhrase** project. Modules attached to PPAT annotate loops that can be refactored into accepted parallel patterns.

4.4 Use case: Detection of pipelines

In this section we present an example of module for PPAT able to detect the pipeline parallel pattern. First, we state the set of attributes defined for annotating pipeline patterns. Next, the concrete internal workings the module part of PPAT are described in detail. Basically, the module works in two phases: *i)* it checks the set of constrains set for the pipeline pattern, and *ii)* it annotates loops that can be refactored into this parallel pattern.

4.4.1 Attributes for annotating parallel patterns

The annotation of pipeline patterns along with their stages is performed using a set of specific C++11 attributes [11]. Specifically, we define a new set of C++11 attributes for defining parallel patterns as annotations in the sequential source codes. In this case, we list the attributes for annotating pipeline parallel patterns:

- `rph::pipeline`: It identifies a loop that can be transformed as a pipeline. This attribute incorporates an argument that identifies the pipeline within the application, so as to know which stages are related to a specific pipeline.
- `rph::stream`: This attribute works together with `rph::pipeline` and identifies the data streams used across stages. Its arguments are the shared variables themselves.
- `rph::stage`: It identifies a code section as a stage of the pipeline, including an argument that unequivocally identifies the stage.
- `rph::plid`: This attribute operates with `rph::stage` and includes an argument that indicates to which pipeline the stage belongs to.
- `rph::in`: This attribute comes along with `rph::stage` and references the input variables passed as arguments.

- `rph::out`: It works along `rph::stage` and references the output variables given as arguments.

Using the aforementioned attributes, a refactorization tool would be able to transform annotated loops into parallel pipelines. Furthermore, they allow to completely separate the detection and refactorization processes, thus different tools can be used in the last step.

4.4.2 Implementation of the pipeline module

In this section we detail internal workings of the pipeline detection module. Specifically, this module is comprised of the three following steps:

1. *Stage detection.* This step is responsible for identifying potential stages in which a loop body can be split. The 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.

   This strategy assumes that each stage has a substantial amount of work, however actual function calls or nested loops inside a stage might have a negligible workload, thus producing unbalanced stages. In the future, we plan to improve this strategy by computing an estimation of the instructions inside each stage, so that the workload between stages can be balanced in a more efficient way.

2. *Checking the pipeline requirements.* This step is responsible for checking a series of requirements that determine whether a loop along with its stages can be refactored into a parallel pipeline. These requirements may be described as stated in Sec. 3.2.2:

   (a) *No global variables modified.* There should not exist instructions that modify global variables in the loop.

   (b) *No RAW dependencies*. There should not exist RAW dependencies of variables used simultaneously within stages of the pipeline.

   (c) *Multiple stages*. The loop should be able to be divided in more than one stage.

   (d) *No break statements*. There should not exist break statements (i.e. `continue`, `break` or `return`) in the loop, as they cannot be paralelized. However, they can be allowed in inner scopes of the main loop.

   The pipeline detection approach guarantees that every single pipeline detected in the source code meets all the aforementioned requirements. Any

other pipeline that does not comply with any of them is automatically discarded by the tool.

3. *Annotation.* The last step the tool is responsible for annotating pipelines found. We leverage the information collected during the preceding step in order to print out attributes on the actual pipelines. Loops discarded by the process are commented with a message that tells the user the reason why it was not considered to be pipeline.

4.5 Evaluation

In this section we perform an experimental evaluation of PPAT using a series of sequential scientific benchmarks in order to analyze how many loops can be refactored into parallel pipelines, using the module described in the previous section. 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.

- *Benchmarks.* To evaluate PPAT, we used the sequential versions of the following set of scientific benchmarks: PARBOIL [25], RODINIA [24], NAS Parallel Benchmarks (NPB) [3] and BIOPERF [2].

The evaluation methodology is based on a comparison between a manual inspection and an automatic one, using PPAT, of the loops appearing in the benchmark codes. To conduct a double-blind study, the manual inspection is performed before the automatic one, so that the manual results are not biased by those from PPAT. For each benchmark analyzed, we collect the number of total lines and loops found, number of pipelines detected (manually and automatically) and a ratio between these two last metrics. So, if this ratio greater than 1, it indicates that PPAT found more pipelines than the human inspection, while if it is lower than 1, it stands for the opposite.

Finally, we discuss the results collected during the manual inspection with those obtained by PPAT in order to demonstrate the quality of the pipeline detection process. In the following, we analyze the results obtained for each benchmark evaluated.

Table 4.1: Statistics for the evopro benchmarks.

Test	Lines	Loops	PPAT	Manual
`ConfigManager.cpp`	1143	34	2	0
`MeasProcessor.cpp`	2060	79	6	3
`RCBProc.cpp`	123	7	1	2
`mainStuff.cpp`	188	11	0	2

4.5.1 Results for evopro benchmark

PPAT tool was tested on the main code base of evopro eRDM use-case. PPAT ran successfully with only minor errors in some renaming and the result is summarised in Table 4.1.

Table 4.1 shows only those files where manual parallelisation was reasonable and the comparison is valuable. As can be seen, the overlap between the automatic pipeline localisation and the manual parallelisation effort is not consistent: sometimes the tool is too eager, sometime it is not effective enough to find all the potential places of pipelines.

Regarding the efficiency of the tool, for example in RCBProc.cpp in Listing 4.3, the pipeline was correctly localized, but its second stage could have been further split in line 120.

```
 99 void RCBProc::decimate(std::vector<int>& out, int start, int len) {
100     static const int newlen = 64;
101     int i;
102     float step = 1.0*(len-1)/(newlen-1);
103     float pos, w1, w2, f;
104     int p1, p2;
105     out.clear();
106     // Generate /newlen/ values
107     [[rph::pipeline, rph::id(0) , rph::stream(pos,p1,out,p2,f,w1,w2)]]
108     for (i = 0; i < newlen; ++i) {
109
110
111         [[rph::stage(0), rph::pipelineid(0), rph::in(i), rph::out(pos,p1)]]
112         {
113             pos = start + i * step;
114         p1 = floor(pos);
115         }
116
117         [[rph::stage(1), rph::pipelineid(0), rph::in(pos,p1,out), rph::out(p2,f,w1,w2,out)]]
118         {
119
120         p2 = ceil(pos);
121         if (p1 == p2)
122             f = filteredData[p1];
123         else {
124             w1 = p2-pos;
125             w2 = pos-p1;
126             f = filteredData[p1]*w1 + filteredData[p2]*w2;
127         }
128         out.push_back(massFactor*f);
129
130         }
131     }
132 }
```

Figure 4.3: PPAT annotations for a `RCBProc.cpp` snippet.

Although in `MeasProcessor.cpp`, shown in Listing 4.4, more than enough pipelines have been identified, some of the computationally expensive parts of the

code have been disregarded by PPAT. For example, between line 1404 and 1407 the inner loop could be pipelined by two separate stages. Maybe due to the encompassing if statement the opportunity has not been notified.

```
1380 void MeasProcessor::processCurves() {
1381
1382     unsigned int axleIdx, measIdx;
1383
1384     //iterate over each assigned measurement entity of every axle
1385     //It isn't a Pipeline
1386      //Less than 2 stages
1387
1388     for (axleIdx = 0; axleIdx < axleData.size(); axleIdx++) {
1389         //It isn't a Pipeline
1390          //Not enough compute
1391         //Less than 2 stages
1392
1393         for (measIdx = 0; measIdx < axleData[axleIdx].axleMeasEnts.size(); measIdx++) {
1394             if (axleData[axleIdx].axleMeasEnts[measIdx] != NULL) {
1395
1396                 //choose the reference curve for the measurement entity
1397                 selectReference(axleIdx, measIdx);
1398
1399                 //get converted curve of current entity
1400                 vector<int16_t> currSamples = axleData[axleIdx].axleMeasEnts[measIdx]->calcConvSamples();
1401                 vector<int> fittedCurve;
1402
1403                 //perform curve fitting for current measurement entity
1404                 fitReference(*(axleData[axleIdx].axleMeasEnts[measIdx]), currSamples, fittedCurve);
1405
1406                 //perform curve evaluation for current measurement entity
1407                 evalCurve(*(axleData[axleIdx].axleMeasEnts[measIdx]), currSamples, fittedCurve);
1408             }
1409         }
1410
1411         //determine if measurement entities can be used for load calculation per-axle
1412         setIsUsable(axleData[axleIdx]);
1413     }
1414 }
```

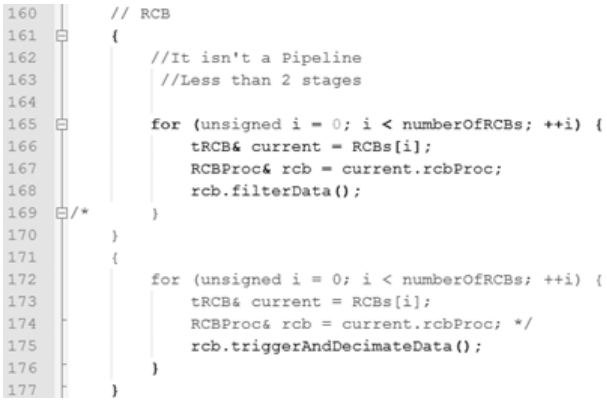
Figure 4.4: Missing PPAT annotations for a `MeasProcessor.cpp` snippet.

In a similar fashon, Listing 4.5 illustrates that in file `mainStuff.cpp`, another pipeline could not be identified by the tool for the for loop in line 165; however it does represent some heavy computation within the application. Although the loop iterations can be executed in parallel and even some minor manual refactoring has been performed (the section between lines 169-174 was commented out in order to merge the two for loops and to increase the number of possible pipeline stages), the result remained the same.

In summary, PPAT usually finds valid pipelines and annotates them properly. However, the pipelines found in our evaluation do not correspond to the places in code that require a lot of computation. In other words, PPAT treat each candidate loops equally, hence the introduction of performance related metric may be needed for its practical industry usage in order to locate real parallelisation hot-spots in code.

4.5.2 Results for the PARBOIL benchmark

The first benchmark used to test PPAT is the PARBOIL benchmark suite, and denoted as PARBOIL. We employ the sequential C codes for each test from the PARBOIL suite in order to detect potential pipelines. Table 4.2 presents the results obtained for the PARBOIL tests. As shown in the table, the number of pipelines found



```
160     // RCB
161     {
162         //It isn't a Pipeline
163          //Less than 2 stages
164
165         for (unsigned i = 0; i < numberOfRCBs; ++i) {
166             tRCB& current = RCBs[i];
167             RCBProc& rcb = current.rcbProc;
168             rcb.filterData();
169 /*      }
170     }
171     {
172         for (unsigned i = 0; i < numberOfRCBs; ++i) {
173             tRCB& current = RCBs[i];
174             RCBProc& rcb = current.rcbProc; */
175             rcb.triggerAndDecimateData();
176         }
177     }
```

Figure 4.5: Missing PPAT annotations for a `mainStuff.cpp` snippet.

from both manual and PPAT inspection are perfectly matching, thus the pipeline detection quality of PPAT is equal to that obtained manually. On the foregoing, we perform a deeper analysis of these results.

Table 4.2: Statistics for the PARBOIL benchmarks.

Test	Lines	Loops	PPAT	Manual	Ratio
`bfs`	237	6	0	0	-
`cutcp`	798	19	0	0	-
`histo`	231	5	0	0	-
`sad`	643	37	1	1	1
`sgemm`	218	8	0	0	-
`spmv`	216	3	0	0	-
`stencil`	229	7	0	0	-
`tpacf`	383	8	0	0	-

We start with the `bfs` test. For this special case, PPAT did not find any pipeline: it discarded the potential ones, since it is not possible to ensure whether they are real or not. The reason is that PPAT cannot check at compile time if there are loop carried dependencies between iterations. In other words, if there have been accesses to a specific position of an array from two different iterations. In this cases, the tool uses a conservative approach and comments the loop with the specific reason that prevented PPAT from annotating it.

Listing 4.1 illustrates an example of a pipeline that PPAT was not able detect due to potential dependencies between iterations, only met at runtime. Analyzing line 6, `index` receives the value from the function `wavefront.front()`, that is only known at run time. Afterwards, in line 23, this value is used for accessing the array `color`, so it cannot be guaranteed that there are no conflicts between iterations. As it can be seen, PPAT printed a comment above the `while` statement (lines 2–4) specifying the arrays that caused such issues.

Next, we focus on the `sad` test. In this case, the code contains a pipeline consisting of 7 stages that passes the data stream from one stage to the next, thus

Listing 4.1: PPAT annotations for a `bfs` snippet.

```
//It isn't a Pipeline
//Feedback: color
//Feedback: color
//Feedback: h_cost
while(!wavefront.empty()){
  index = wavefront.front();
  wavefront.pop_front();
  //It isn't a Pipeline
  //Feedback: color
  //Feedback: h_cost
  //Less than 2 stages
  for(int i=h_graph_nodes[index].x;
    i<(h_graph_nodes[index].y +
    h_graph_nodes[index].x); i++)
    {
      int id = h_graph_edges[i].x;
      if(color[id] == WHITE){
        h_cost[id]=h_cost[index]+1;
        wavefront.push_back(id);
        color[id] = GRAY;
      }
    }
  color[index] = BLACK;
}
```

meeting the requirements stated in Section 4.4. An issue that we found for this test is that it internally uses C macros. Thus, Clang frontend receives a code from the preprocessor in which all macros have been replaced by their corresponding values, possibly having shifted lines with respect to the original code. Because of that, PPAT it is not able to insert comments/annotations on the proper locations. The problem however, comes from the Clang functions that obtain the code locations, that work with respect to the original code and not with the preprocessed one. To bypass this issue, we manually replace macros in the source code by their corresponding values. Being aware that this is not the best solution, we plan to address this issue in a future version of PPAT.

The rest of tests analyzed do not present any pipeline. Nevertheless, they contain other parallel patterns such as *reduce*, *farm* or *stencil*.

4.5.3 Results for the RODINIA benchmark

The second benchmark tested is RODINIA. Again, we leverage the sequential versions of the tests contained in the RODINIA benchmark suite to evaluate PPAT. Table 4.3 shows the results obtained for this benchmark. As can be seen, there are only two tests in which PPAT found potential pipelines: `cfd` and `mummergpu`. In other tests, neither PPAT nor us find pipeline patterns. However, we observe farms as the major parallel patterns encountered within the source codes.

As an example, Listing 4.2 presents an annotated pipeline by PPAT found in the `cfd` test. As can be seen, the pipeline is parallelizable, since it meets all re-

Table 4.3: Statistics for the RODINIA benchmarks.

Test	Lines	Loops	PPAT	Manual	Ratio
backsprop	764	28	0	0	-
bfs	197	7	0	0	-
b+tree	2,768	80	0	0	-
cfd	2,598	78	4	4	1
heartwall	1,757	54	0	0	-
kmeans	1,806	20	0	0	-
lavaMD	376	10	0	0	-
mummergpu	8,468	44	1	1	1
nn	234	8	0	0	-
nw	292	12	0	0	-
particlefilter	773	44	0	0	-
streamcluster	2,996	48	0	0	-

quirements stated in Section 4.4. Note that PPAT has included RePhrase attributes accordingly, thus representing stages and other parameters related to the pipeline.

4.5.4 Results for the NAS benchmark

The NAS benchmark is the next one used to evaluate PPAT. This benchmark suite is comprised of a set tests written in C. Table 4.4 shows the results obtained by PPAT. As can be seen, PPAT finds several cases in which there are potential pipelines. In most cases, PPAT finds the same number of pipelines that the manual analysis, however there are other cases in which PPAT is not able to detect some of them, e.g., for the `FT` test. This is due to the loops contain dynamic dependencies that PPAT cannot meet statically. On the contrary, for some other cases PPAT finds pipelines where the human inspection cannot, e.g., in `DC` and `UA` tests.

Table 4.4: Statistics for the NAS benchmarks.

Test	Lines	Loops	PPAT	Manual	Ratio
CG	5,374	45	1	1	1
DC	3,227	104	2	1	2
EP	324	8	0	0	-
FT	1,056	41	2	3	0.6
IS	856	16	0	0	-
MG	1,547	80	1	1	1
SP	4,251	250	1	1	1
UA	9,677	478	3	2	1.5

Listing 4.3 shows an example of code from the `EP` test in which PPAT has discarded the loop to be parallelized using a concurrent pipeline pattern. This is due to two different reasons. First, this loop presents feedback on the different variables (`sy`, `sx` and `q` in lines from 15 to 17), thus violates requirement 2b. On the other hand, a global value is being written and read, therefore producing side effects and violating requirement 2a. As can be seen in line 15, the global vector `q` is accessed for read and write on the same position `l`. In the end, since PPAT

Listing 4.2: PPAT annotations for a `cfd` snippet.

```
[[ rph::pipeline(0) , rph::stream(density, density_energy, velocity, speed_sqd,
   (cont.)pressure) ]]
for(int i = 0; i < nelr; i++)
{
    float density, density_energy, speed_sqd, pressure, speed_of_sound;
    float3 velocity, momentum;
    [[ rph::stage(0), rph::plid(0), rph::out(density, density_energy, velocity) ]]
    {
        density = variables[NVAR*i + VAR_DENSITY];
        momentum.x = variables[NVAR*i + (VAR_MOMENTUM+0)];
        momentum.y = variables[NVAR*i + (VAR_MOMENTUM+1)];
        momentum.z = variables[NVAR*i + (VAR_MOMENTUM+2)];
        density_energy = variables[NVAR*i + VAR_DENSITY_ENERGY];
        compute_velocity(density, momentum, velocity);
    }
    [[ rph::stage(1), rph::plid(0), rph::in(velocity), rph::out(speed_sqd) ]]
    speed_sqd = compute_speed_sqd(velocity);

    [[ rph::stage(2), rph::plid(0), rph::in(density, density_energy, speed_sqd), rph
       (cont.)::out(pressure) ]]
    pressure = compute_pressure(density, density_energy, speed_sqd);

    [[ rph::stage(3), rph::plid(0), rph::in(density, pressure), rph::out(
       (cont.)speed_of_sound) ]]
    speed_of_sound = compute_speed_of_sound(density, pressure);

    [[ rph::stage(4), rph::plid(0), rph::in(speed_sqd, speed_of_sound) ]]
    step_factors[i] = float(0.5f) / (std::sqrt(areas[i]) * (std::sqrt(speed_sqd) +
       (cont.)speed_of_sound));
}
```

cannot ensure if these accesses are iteration-independent, it decides not to consider the loop as a parallel pipeline due to possible side effects.

For the `IS` test, PPAT does not detect any pipeline because all loops defined for this test have not enough compute to perform or do not have, at least, two stages. Therefore, it violates requirement 2c. For instance, Listing 4.4 shows a loop that PPAT discarded due to it cannot be divided in two stages.

4.5.5 Results for the BIOPERF benchmark

The last set of test used to evaluate PPAT is the BIOPERF, a benchmark suite comprised of C tests to evaluate high-performance computer architecture on bioinformatics applications. Table 4.5 presents the results obtained by both PPAT and manual inspections of the code. As can be seen, the tool detects more pipelines than human inspection for the `clustalw_smp` and `Hmmer` tests. Indeed, the ratios obtained for these benchmarks are greater than 1. This is due to source codes from BIOPERF (in the range of thousands of lines) that are much larger than in other tests and, because of that, the detection of pipelines just by visual inspection becomes a much more complicated and error-prone task. We believe that, in such cases, a tool like PPAT can tremendously aid developers in detecting such parallel patterns.

Listing 4.3: PPAT annotations for a loop from the `EP` test.

```
//It isn't a Pipeline
//Feedback: sy
//Feedback: sx
//Feedback: q
//GLOBAL VALUE USED
for (i = 0; i < NK; i++) {
  x1 = 2.0 * x[2*i] - 1.0;
  x2 = 2.0 * x[2*i+1] - 1.0;
  t1 = x1 * x1 + x2 * x2;
  if (t1 <= 1.0) {
    t2   = sqrt(-2.0 * log(t1) / t1);
    t3   = (x1 * t2);
    t4   = (x2 * t2);
    l    = MAX(fabs(t3), fabs(t4));
    q[l] = q[l] + 1.0;
    sx   = sx + t3;
    sy   = sy + t4;
  }
}
```

Listing 4.4: PPAT annotations for a loop from the `IS` test.

```
//It isn't a Pipeline
//Less than 2 stages
for (i=0; i<NUM_KEYS; i++)
{
  x = randlc(&seed, &a);
  x += randlc(&seed, &a);
  x += randlc(&seed, &a);
  x += randlc(&seed, &a);
  key_array[i] = k*x;
}
```

Table 4.5: Statistics for the BIOPERF benchmarks.

Test	Lines	Loops	PPAT	Manual	Ratio
`CE`	4,316	168	3	3	1
`clustalw_smp`	28,428	929	14	8	1.75
`Hmmer`	26,153	423	12	11	1.09

5. Conclusions and future works

In this deliverable, we have reviewed the state-of-the-art about and existing program shaping and pattern detection tools and described the set of program shaping techniques and methods in order to refactor sequential C++ programs into hygienic C++ code with equivalent functionality. Next, we have described for each pattern of the initial set of **RePhrase** along the conditions and requirements that need to be satisfied in order to be introduced in C++ applications. Finally, we presented a prototype pattern discovery tool that instances of pipeline pattern candidates within C++ applications at compile time.

A tool contribution presented in this deliverable is the Parallel Pattern Analyzer Tool (PPAT), as a prototype tool for discovering parallel patterns in sequential applications, as proposed in the WP2 in the **RePhrase** project. PPAT has several features: It is completely independent of the refactoring tool used since it identifies parallel patterns. Furthermore, it performs a static analysis and avoids the use of profiling techniques and guarantees that parallel patterns detected comply with a series of requirements that prove the correctness of the solution. As a future work, we plan to extend the Parallel Pattern Analyzer tool to include patterns from the advanced set of patterns from the **RePhrase** project. Furthermore, we will add support in order 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] D.A. Bader, Yue Li, Tao Li, and V. Sachdeva. BioPerf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 163–173, Oct 2005.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[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] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, and Tino Breddin. Paraphrasing: Generating parallel programs using refactoring. In Bernhard Beckert, Ferruccio Damiani, FrankS. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 237–256. Springer Berlin Heidelberg, 2013.

[6] Cilk Plus home page, 2016. https://www.cilkplus.org/.

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

[8] Danny Dig. A refactoring approach to parallelism. *IEEE Softw.*, 28(1):17–22, January 2011.

[9] FastFlow home page, 2016. http://calvados.di.unipi.it/fastflow.

[10] Zia Ul Huda, Ali Jannesari, and Felix Wolf. Using template matching to infer parallel design patterns. *ACM Trans. Archit. Code Optim.*, 11(4):64:1–64:21, January 2015.

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

[12] Ralf Jahr, Mike Gerdes, and Theo Ungerer. A pattern-supported parallelization approach. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 53–62, New York, NY, USA, 2013. ACM.

[13] 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.

[14] 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.

[15] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Reengineering for parallelism: An entry point into plpp for legacy applications: Research articles. *Concurr. Comput.: Pract. Exper.*, 19(4):503–529, March 2007.

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

[17] Anne Meade, Jim Buckley, and J. J. Collins. Challenges of evolving sequential to parallel code: An exploratory review. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 1–5, New York, NY, USA, 2011. ACM.

[18] 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.

[19] 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.

[20] 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.

[21] OpenMP home page, 2016. http://openmp.org/wp/.

[22] 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.

[23] 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.

[24] C. Shuai, M. Boyer, M. Jiayuan, D. Tarjan, J. W. Sheaffer, L. Sang-Ha, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[25] J. A. Stratton, C. Rodrigues, I-J. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. IMPACT Technical Report. http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf, 2012.

[26] H. Sutter. Welcome to the Jungle. http://herbsutter.com/welcome-to-the-jungle/, 2012. [Last access 6th May 2015].

[27] INTEL Thread Building Blocks home page, 2016. https://www.threadingbuildingblocks.org/.

[28] 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.