Project no. 644235

REPHRASE

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

Software for the Initial Refactoring Tool D2.2

Due date of deliverable: January 2016

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

Responsible institution: University of St Andrews
Editor and editor's address: Chris Brown, University of St Andrews

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

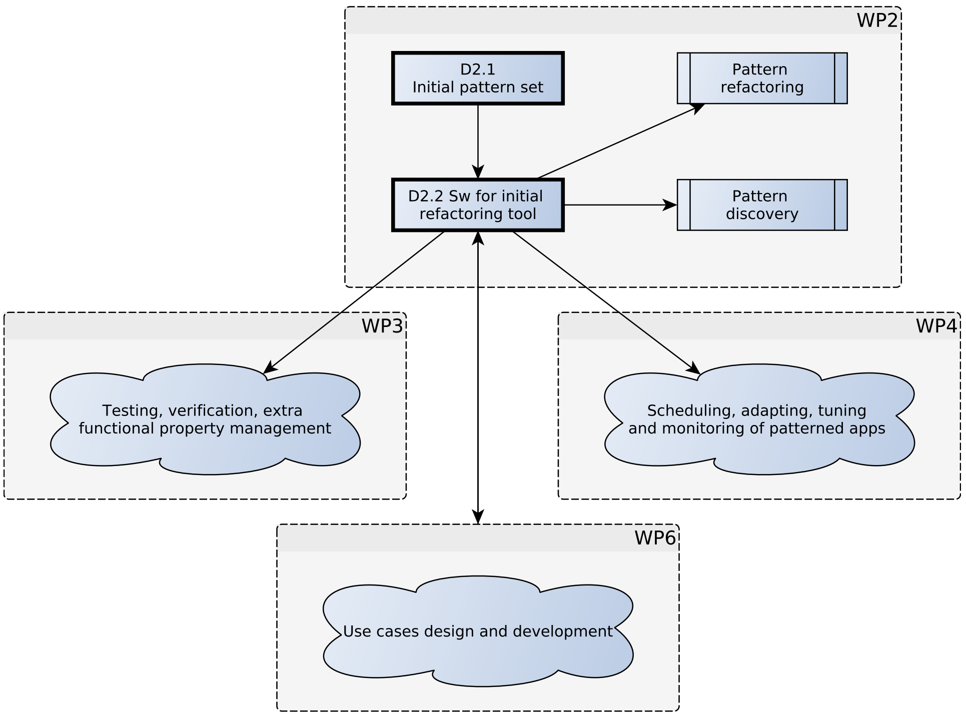
Rev.	Date	Who	Site	What
1	31/01/16	Christopher Brown	USTAN	Original Version
2	11/10/16	Kevin Hammond	USTAN	Review and corrections (grammar, typos, formulation)
3	20/02/17	Marco Danelutto	UNIPI	Added placement picture in executive summary

Contributions Per Partner

Institution	**Contribution**
USTAN	Refactoring tool (Sections 2 and 4), contribution to the RPL and experiments (Sections 3 and 5)
UNIPI	Contribution to the RPL and experiments (Sections 3 and 5)
UNITO	Contribution to the RPL and experiments (Sections 3 and 5)
UC3M	Contribution to the RPL and experiments (Sections 3 and 5)

Executive Summary

Parallelising sequential applications is usually a very hard job, due to many different ways in which an application can be parallelised and a large number of programming models (each with its own advantages and disadvantages) that can be used. In this deliverable, we describe a method to semi-automatically generate and evaluate different parallelisations of the same application, allowing programmers to find the best parallelisation without significant manual reengineering of the code. We describe a novel, high-level domain-specific language, *Refactoring Pattern Language (RPL)*, that is used to represent the parallel structure of an application and to capture its extra-functional properties (such as service time). We then describe a set of RPL rewrite rules that can be used to generate alternative, but semantically equivalent, parallel structures (parallelisations) of the same application. We also describe the *RPL Shell* that can be used to evaluate these parallelisations, in terms of the desired extra-functional properties. Finally, we describe a set of C++ refactorings, targeting OpenMP, Intel TBB and FastFlow parallel programming models, that semi-automatically apply the desired parallelisation to the application's source code, therefore giving a parallel version of the code. We demonstrate how the RPL and the refactoring rules can be used to derive efficient parallelisations of two realistic C++ use cases (Image Convolution and Ant Colony Optimisation).


WP2
D2.1
Initial pattern set
Pattern
refactoring
D2.2 Sw for initial
refactoring tool
Pattern
discovery
WP3
Testing, verification, extra
functional property management
WP4
Scheduling, adapting, tuning
and monitoring of patterned apps
WP6
Use cases design and development

Contents

Chapter 1

Introduction

Despite the emergence of multi-core and many-core systems, parallel programming is still a very laborious task that is difficult to get right. Most current application designers and programmers are not experts in writing parallel programs. Knowing *where* and *when* to introduce parallel constructs, as well as *what* construct to introduce, can be a daunting and seemingly *ad-hoc* process. As a result, parallelism is often introduced using an abundance of low-level concurrency primitives, such as explicit threading mechanisms and communication, which typically do not scale well and can lead to deadlock, race conditions etc. Furthermore, software engineering tasks such as porting to other parallel platforms and general code maintenance can then require huge efforts and often rely on good parallel systems expertise.

Parallel patterns attempt to hide away this complexity by providing parameterised implementations of common types of parallel operations, such as parallel *farm* (a.k.a. *parallel-for*), *pipeline* and *reduce*. While making the task of parallel programming easier, the programmer still needs to choose the appropriate pattern *structure* for their problem and instantiate the patterns properly; this is a highly complex process. In addition, there is a wide choice of pattern libraries for C++, with over 10 libraries available at the time of writing. Moreover, each of these pattern libraries offers different benefits and limitations, including different sets of supported skeletons and primitives, and different complexities of writing parallel code. Porting sequential C++ code to use one of these libraries and then porting the parallel code to an alternative pattern library still requires very significant effort.

In this deliverable (illustrated in Figure 1.1) we exploit *refactoring* tool-support to offer semi-automatic program transformations that, i) transform sequential code into its parallel equivalent, by introducing all of the complex parallel implementation via skeleton libraries; and, ii) transform a parallel implementation written in one skeleton library into an alternative implementation using a different library. We also describe a new high-level domain-specific language, *Refactoring Pattern Language (RPL)*, for describing the abstract and high-level parallel structure of a C++ application, together with a system of *rewrite rules* that operate over the RPL expressions, allowing transformations between equivalent parallel structures

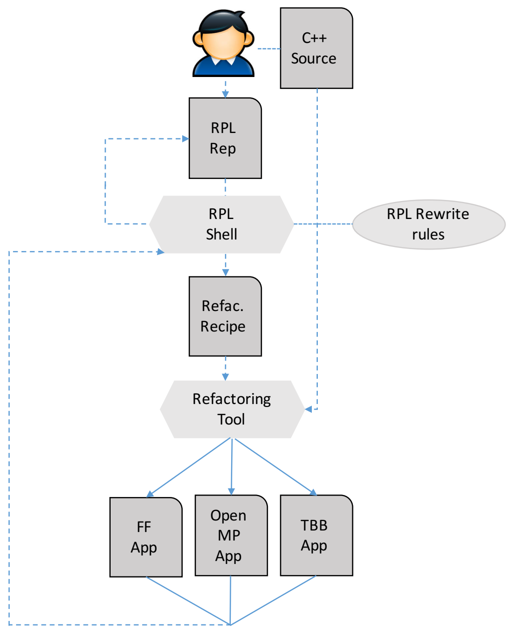


Figure 1.1: A process of parallelising C++ applications using RPL and refactoring.

of the same application. Furthermore, the rewrite system also provides runtime estimations, guiding the user to choose the optimal parallel structure. Finally, once an optimal structure is obtained, the rewrite system returns a *refactoring recipe*: a report that shows the steps necessary for the user to then refactor their application using tool-support in order to introduce the correct skeletons and nestings to obtain this optimal configuration. This allows programmers to write, with minimal effort, performance-portable parallel applications and to adapt them easily to new architectures by changing between different parallel structures and implementations. Finally, we evaluate our approach by parallelising two realistic C++ applications, Image Convolution and Ant Colony Optimisation, demonstrating that we can achieve scalable speedups with small programming effort. To show generality, our approach is demonstrated on three parallel frameworks – FastFlow [1], Intel Threading Building Blocks (TBB [6]) and OpenMP [3].

Chapter 2

The Refactoring Pattern language (RPL)

The *Refactoring Pattern Language* (RPL) allows application programmers to design patterned applications without having to deal with low-level implementation details. RPL provides a number of different features, including:

i) *efficient and concise representation of the full set of patterns used to express the parallel structure of applications*, allowing the application programmer to abstract the parallelism exploitation strategies from the actual code and to implement these strategies in terms of the available library mechanisms;

ii) *support for pattern rewriting techniques, together with mechanisms to estimate the non-functional properties exposed by patterns*, allowing the application programmer to experiment with different patterns/pattern compositions at design time, right before going to the actual coding phase;

RPL has been designed as a high-level DSL, independent of the underlying programming language or tools used to implement the parallel applications. It provides the programmers with the ability to represent **pattern expressions**. RPL currently supports a limited set of stream- and data-parallel patterns, including those discussed in Deliverable D2.1, and supports the definition of pattern expressions via the following (simplified) grammar:

$$\begin{aligned} \mathsf{Pat} ::= \; & \mathsf{seq} \mid \mathsf{streamGen} \mid \mathsf{streamDrain} \mid \mathsf{pipe}(\mathsf{Pat}, \mathsf{Pat}) \mid \\ & \mathsf{comp}(\mathsf{Pat}, \mathsf{Pat}) \mid \mathsf{farm}(\mathsf{Pat}) \mid \mathsf{map}(\mathsf{Pat}) \end{aligned} \tag{2.1}$$

The seq pattern represents a sequential code wrapped into a function that accepts inputs of type T_{IN} and produces the results of type T_{OUT}. streamGen and streamDrain represent sequential patterns producing and consuming a stream of some data type T_{STREAM}. comp represents a sequential composition of two patterns, where first all outputs for the first pattern are produced and then sent as an input to the second pattern. pipe,farm and map represent pipeline, farm and map patterns, as described in Deliverable D2.1. As an example, a video processing application applying a set of filters to video frames may be expressed in RPL as

Rule	*Name*
seq (P) $\rightarrow$ farm (P)	farm_intro
farm (P) $\rightarrow$ P	farm_elim
comp (P_1, P_2) $\rightarrow$ pipe (P_1, P_2)	pipe_intro
pipe (P_1, P_2) $\rightarrow$ comp (P_1, P_2)	pipe_elim
comp (map (P_1), map (P_2)) $\rightarrow$ map (comp (P_1, P_2))	map_prom
map (P) $\rightarrow$ comp (streamGen,farm (P),streamDrain)	data_strm

Figure 2.1: Pattern rewriting rules

pipe(streamGen, seq(*filt*$_1$), . . . , seq(*filt*$_n$), streamDrain).

Each term in a pattern expression can have one or more **attributes** (which represent different extra-functional properties of the pattern) associated with it. We will denote this by Pat$\leftarrow$$attribute$. For example, we say that the term has a *parallelism degree* of n (denoted by NW(n)) if n units of work (e.g. operating system threads) are used in parallel for the execution of that term. We can compute the parallelism degree of a pattern expression using the following simple rules:

- seq$\leftarrow$NW(1);
- parallel degrees of farm and map patterns are assigned by a programmer (or programming tool);
- if Pat1$\leftarrow$NW(m) and Pat2$\leftarrow$NW(n), then comp (Pat1,Pat2)$\leftarrow$NW$(\max(m, n))$;
- if Pat1$\leftarrow$NW(m) and Pat2$\leftarrow$NW(n), then pipe (Pat1,Pat2)$\leftarrow$NW$(m + n)$;

As another example, *service time* can be used in streaming applications as a performance measure. The service time of a stream pattern expression may be computed from the service times of its sequential components and of the parallelism degrees in the different patterns appearing in the expression. Therefore, provided that we have obtained the service times of the seq nodes (e.g. using profiling), we can automatically calculate the service time of the whole pattern expression. We can use this information to compare different parallelisations and estimate which of them will give the best performance. We note that RPL focuses purely on specifying and reasoning about patterns, and, as such, we demonstrate RPL here using simple models for the sake of conciseness.

We have implemented a prototype of the *RPL shell* that provides the main features of the RPL described in this deliverable. The shell contains a set of well-known *pattern rewriting rules* (see Figure 3.1) that are used to generate alternative pattern expressions with the same functional semantics, but possibly different extra-functional properties. Below is a typical session the application programmer may run:

1. *Represent application parallel structure through RPL.* Suppose the application we want to represent is a simple pipeline with two stages, where the first

stage processes an item in k time units and the second stage processes an item in $3k$ time units. The programmer may represent the parallel structure of his/her application as follows:

```
# let stage1 = Seq("f");;
# let stage2 = Seq("g");;
# let app = Pipe(stage1,stage2);;
# pp app;;
pipe
  seq f
  seq g
```

2. *Annotate pattern expressions with attributes*. The user can provide the service times of the seq terms.

```
# let stage1 = Aseq("f",[Ts(4.0)]);;
# let stage2 = Aseq("g",[Ts(12.0)]);;
# let app1 = Apipe(stage1,stage2,[]);;
# ppa app1;;
pipe
   seq f with Ts(4.000000)
   seq g with Ts(12.000000)
```

3. *Evaluate extra-functional properties of the pattern expression*. The application programmer may then compute the attributes (in this case, the degree of parallelism and service time) for the top-level pattern expression, app1:

```
# servicetime app1;;
- : float = 16.
# pardegree app1;;
- : int = 2
```

4. *Refactor pattern expressions*. RPL supports refactoring of pattern expression to pattern expressions according to well know, functional semantics preserving rules. An extensible set of rules is associated to RPL modelling these rules. The user interacting through the RPL shell may apply transformations coded in the rules to the pattern expression at hand, the one modelling the parallel behaviour of his/her application, such that alternative, functionally equivalent[1] parallel application structures may be automatically derived. Here, p4 represents a *pipeline* pattern, where p5 represents a sequential composition.

```
# let p1 = Seq("f");;
val p1 : patterns = Seq "f"
# let p2 = Seq("g");;
```

[1]given a couple of pattern expressions, we say they are *functionally equivalent* iff they compute the same results out of the same inputs, irrespectively of the associated non functional properties (e.g. performance, efficiency, security, etc.)

```
val p2 : patterns = Seq "g"
# let p3 = Pipe(p1,p2);;
val p3 : patterns = Pipe (Seq "f", Seq "g")
# let p4 = pipe_elim(p3);;
val p4 : patterns = Comp (Seq "f", Seq "g")
# let p5 = farm_intro(p4);;
val p5 : patterns = Farm (Comp (Seq "f", Seq "g"))
#
```

5. *Evaluate alternative pattern expressions*. The application programmer, unsatisfied with the extra-functional features of his/her pattern expression, may apply some rewriting rules and ask for the properties of the new expression:

```
# let app2 = rr_pipe_elim(app1);;
# ppa app2;;
comp
  seq f with Ts(4.000000)
  seq g with Ts(12.000000)
# servicetime app2;;
- : float = 16.
# let app3 = rr_farm_intro app2 10;;
# ppa app3;;
farm with Pd(10) Te(1.000000)
  comp
    seq f with Ts(4.000000)
    seq g with Ts(12.000000)
# servicetime app3;;
- : float = 1.6
```

   It is worth pointing out that a completely different pattern expression could have been derived that leads to a comparable service time, but with different total parallelism degree (resources needed):

```
# let farm1 = rr_farm_intro stage1 3;;
# let farm2 = rr_farm_intro stage2 8;;
# let app4 = Apipe(farm1, farm2, []);;
# ppa app4;;
pipe
  farm with Pd(3) Te(1.000000)
    seq f with Ts(4.000000)
  farm with Pd(8) Te(1.000000)
    seq g with Ts(12.000000)
# servicetime app4;;
- : float = 1.5
# pardegree app3;;
- : int = 10
# pardegree app4;;
- : int = 11
```

6. *Generating a refactoring recipe* Once the programmer is satisfied with the extra-functional properties of the derived pattern expression, he/she can generate a *refactoring recipe* that contains the steps needed to transform the actual C++ application into its parallel version with the desired parallel structure. This recipe can then be used by the refactoring tool (Section 4) to

parallelise the code. See Section 5 for example refactoring recipes. A *refactoring recipe* is a finite sequence of deterministic refactoring steps that eventually produces the recipe result out of an initial pattern expression. Fig. 3.2 outlines an example of refactoring reciped derived through interaction with RPL.

7. *Pattern expression optimizers* We also have a prototype implementation of some pattern expression optimizers, that automate the process of generating alternative parallel structures and optimising extra-functional properties of interest.

```
# let app_opt1 = ts_optim app1;;
# ppa app_opt1;;
pipe
  seq f with Ts(4.000000)
  farm with Pd(3)
    seq g with Ts(12.000000)
- : unit = ()
# servicetime app_opt1;;
- : float = 4.
```

The `ts_optim` rewriter in this case farms out the slower stages of the pipeline, so that the overall service time is reduced to the service time of the slowest stage. A different optimizer computes the normal form of a stream parallel pattern expression:

```
# let ap1 = Aseq("f", [Ts(2.0)]);;
val ap1 : annot_skel = Aseq ("f", [Ts 2.])
# let ap2 = Aseq("g", [Ts(4.0)]);;
val ap2 : annot_skel = Aseq ("g", [Ts 4.])
# let ap3 = Apipe(ap1,ap2,[]);;
val ap3 : annot_skel = Apipe (Aseq ("f", [Ts 2.]), Aseq ("g", [Ts 4.]), [])
# let ap4 = Afarm(ap3,[Pd(5);Te(1.0)]);;
val ap4 : annot_skel =
  Afarm (Apipe (Aseq ("f", [Ts 2.]), Aseq ("g", [Ts 4.]), []), [Pd 5; Te 1.])
# ppa ap4;;
farm with Pd(5) Te(1.000000)
  pipe
    seq f with Ts(2.000000)
    seq g with Ts(4.000000)
- : unit = ()
# let ap7 = rr_normal_form ap4;;
val ap7 : annot_skel =
  Afarm (Acomp (Aseq ("f", [Ts 2.]), Aseq ("g", [Ts 4.]), []),
   [Te 1.])
# ppa ap7;;
farm with Te(1.000000)
  comp
    seq f with Ts(2.000000)
    seq g with Ts(4.000000)
- : unit = ()
#
```

```
# ap4;;                          (* this is the initial pattern expression *)
- : annot_skel =
Afarm (Apipe (Aseq ("f", [Ts 2.]), Aseq ("g", [Ts 4.]), []), [Pd 5; Te 1.])
# pardegree ap4;;              (* ask the pardegree (# of resources needed) *)
- : int = 10
# servicetime ap4;;   (* ask the predicted service time according to annot *)
- : float = 0.8
# let ap4b = rr_pipe_elim (rr_farm_elim ap4);;     (* refactor to diff expr *)
val ap4b : annot_skel = Acomp (Aseq ("f", [Ts 2.]), Aseq ("g", [Ts 4.]), [])
# pardegree ap4b;;                                    (* ask new par degree *)
- : int = 1
# servicetime ap4b;;                            (* and ask new service time *)
- : float = 6.
# let ap4c = (rr_farm_intro ap4b 6) ;;         (* apply further refactoring *)
val ap4c : annot_skel =
  Afarm (Acomp (Aseq ("f", [Ts 2.]), Aseq ("g", [Ts 4.]), []), [Pd 6; Te 1.])
# servicetime ap4c;;                                  (* service time now is? *)
- : float = 1.
# pardegree ap4c;;
- : int = 6           (* that's ok, let's remember the refactoring recipe *)
# let recipe1 = Recipe [rr_farm_elim; rr_pipe_elim; (rr_farm_intro 6)];;
val recipe1 : refactorrecipe = Recipe [<fun>; <fun>; <fun>]
```

Figure 2.2: Sample refactoring recipe derivation

2.1 Installing the RPL proof-of-concept prototype

The proof-of-concept prototype of RPL has only been developed with the aim of demonstrating the feasibility of the RPL concepts and to provide a proof-of-concept to the project partners. This means the proof-of-concept proptotye software has not been developed respecting all the software engineering best practices, nor it is intended, in this version, to be useful to develop actual parallel code or actual parallel refactoring recipes.

The tarball with the software sources is available at the RePhrase web site (`http://rephrase-ict.eu/`) under *Deliverables, D2.2*. The proof-of-concept prototype is entirely written in Ocaml and has been developed and tested under Linux (Ubuntu 15.04, Linux kernel 3.19) with Ocaml version 4.01.0. Therefore to run the proof-of-concept prototype a similar Ocaml system is required[2].

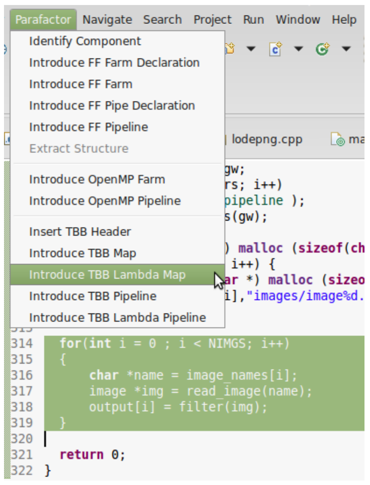
The steps are to be followed to test the software:

1. extract the files from the source archive file `rplPOCP.tgz` using a `tar xvf rplPOCP.tgz`
2. compile everything with the supplied makefile: `make apattern`
3. launch the rpl shell with a command `./rplshell`
4. interact with the shell with the available visitors, refactorers and optimizers. Remember to `open Apattern` to use the annotated pattern grammar, and to `open Arewrite`, `open Avisitor` and `open Aoptim` to use the refactoring rules, the defined visitors and optimizers, respectively.

[2]In principle, the prototype may run under Windows or Mac OS X as well, provided that the suitable version of Ocaml is installed. We have not tested it with these OS, however.

```
marcod@marcod-yoga:~/Documents/Code/DSL/rplPOCP$ ./rplshell
        OCaml version 4.01.0

# open Apattern;;
# let s1 = Aseq("f", [Ts(5.0)]);;
val s1 : Apattern.annot_skel = Aseq ("f", [Ts 5.])
# let s2 = Aseq("g",[Ts(7.0)]);;
val s2 : Apattern.annot_skel = Aseq ("g", [Ts 7.])
# let p = Apipe(s1,s1,[]);;
val p : Apattern.annot_skel =
  Apipe (Aseq ("f", [Ts 5.]), Aseq ("g", [Ts 7.]), [])
# open Avisitor;;
# servicetime p;;
- : float = 5.
# pardegree p;;
- : int = 2
# open Arewrite;;
# rr_farm_intro p;;
- : Apattern.annot_skel =
Afarm (Apipe (Aseq ("f", [Ts 5.]), Aseq ("g", [Ts 7.]), []), [Pd 2; Te 1.])
#
```



```
gw;
rs; i++)
pipeline );
s(gw);

) malloc (sizeof(ch
 i++) {
ar *) malloc (sizeo
i],"images/image%d.

  for(int i = 0 ; i < NIMGS; i++)
  {
      char *name = image_names[i];
      image *img = read_image(name);
      output[i] = filter(img);
  }

  return 0;
}
```

Figure 3.1: Our refactoring tool built into Eclipse, complete with a menu of refactorings for FastFlow, OpenMP and TBB

Chapter 3

Refactorings to Introduce Parallel Patterns

Refactoring is the process of changing the structure of a program, while preserving its functionality [5]. Unlike automatic program compilation and optimisation, refactoring emphasises the software development cycle, principally by: i) improving the design of software; ii) making software easier to understand; and iii) encouraging code reuse. This leads to increased productivity and programmability. Our refactoring tool (Figure 4.1) is implemented in the Eclipse development platform, using the CDT plugin for C++ development. This *semi-automatic* approach is more general than fully automated parallelisation techniques, which typically work only for a very limited set of cases under specific conditions, and are not easily tractable. Furthermore, unlike e.g. simple loop parallelisation, refactoring is applicable to a wide range of possible parallel structures, since parallelism is introduced in a structured way through algorithmic skeletons.

In this section we discuss new refactoring implementations which build upon earlier refactorings that introduce FastFlow skeletons [2]. We have implemented a set of refactorings for both Intel TBB and OpenMP.

We present two refactorings: *introduce map* and *introduce pipeline*. Each of these has two variants, according to the type of *component* involved. A *component* (or *task*) is a piece of code that will be executed in parallel as part of a parallel pattern. Typically components are computationally intensive pieces of code, such as expensive functions.

In TBB, components can be defined in two ways: using either *lambda expressions* or *function objects* (sometimes called *functors* in C++). These two approaches provide different advantages to the user. Lambda expressions (introduced in C++11 ([7, 11.4])) are anonymous functions which are useful for simple operations which are only used at a single point in the program and consist of only two or three statements. If the application requires the same operation in many places, or contains more complex code, functors are preferable: these are classes which overload the function application operator `()`, allowing instances of the class to be applied as if they were functions ([7, 3.4.3]).

```
for(int i = 0 ; i < NIMGS; i++) {
  char *name = image_names[i];
  image *img = read_image(name);
  output[i] = filter (img);
}
```

Figure 3.2: Sequential code before refactoring

We will illustrate our refactorings by showing the results when they are applied to the sequential code in Figure 4.2. This represents a loop which iterates over an array of strings containing names of image files: it reads each image from disk, applies some function called `filter`, and stores the result in the array `output`.

3.1 Parallel Maps

Maps using lambda expressions. This refactoring takes a loop, converts the body into a lambda expression, and applies TBB's `parallel_for` function to execute the iterations of the loop in parallel. If we apply this refactoring to the sequential code in Figure 4.2, the original loop is replaced with the code shown in Figure 4.3.

The result consists of a single expression which applies the `parallel_for` construct to two arguments: a TBB object called a *blocked range* which represents the range of values for which parallel execution is to take place, and a C++ lambda expression encapsulating the original sequential code. The syntax `[&]` at the start of the lambda expression indicates that any free variables in the body of the lambda expression should be captured by reference. It is also possible to use `[=]` to indicate capture-by-value, or `[]` to indicate that no variables are expected to be

```
  tbb::parallel_for(tbb::blocked_range<int>(0, NIMGS),
                 [&](const tbb::blocked_range<int>& r) {
         for (int i = r.begin(); i != r.end(); ++i) {
           char* name = image_names[i];
           image* img = read_image(name);
           output[i] = filter (img);
         } });
```

Figure 3.3: Code after introduction of a TBB lambda map

captured. When the refactoring is initiated, the user is presented with a dialogue where the capture method can be specified, and also the name of the parameter for the blocked range object (`r` in Figure 4.3). The rest of the process is entirely automatic.

Maps using function objects. As an alternative to the lambda-based refactoring shown above, the user can choose to introduce a map based on a function object. In this case, a new class is introduced in the source code immediately before the function containing the code which is being refactored, and the original loop is replaced by a single `parallel_for` statement. The results for our sample code are shown in Figures 4.4 and 4.5.

```
class ProcessImage {
  char** image_names;
  char** output;

public:
  void operator()(const tbb::blocked_range<int>& r) const {
    for (int i = r.begin(); i != r.end(); ++i) {
      char* name = image_names[i];
      image* img = read_image(name);
      output[i] = filter (img);
    }
  }

  ProcessImage(char** arg_image_names, char** arg_output) :
  image_names(arg_image_names), output(arg_output) { }
};
```

Figure 3.4: C++ function object definition encapsulating sequential code

```
tbb::parallel_for(tbb::blocked_range<int>(0, NIMGS),
                  ProcessImage(image_names, output));
```

Figure 3.5: Invocation of TBB `parallel_for` statement to apply function object

As with the lambda map refactoring, the user is prompted for certain parameters (the name of the functor class and the name of the blocked range, `ProcessImage` and `r` in this case), but the rest of the process is automatic. The

```
1  tbb:: filter_t  <void, char*> stage_1(tbb::filter:: serial_in_order,
2                                        Closure_1(image_names));
3  tbb:: filter_t  <char*, image*> stage_2(tbb::filter::serial,  Closure_2());
4  tbb:: filter_t  <image*, void> stage_3(tbb::filter::serial_in_order,
5                                        Closure_3(output));
6  tbb:: parallel_pipeline(16,  stage_1 & stage_2 & stage_3);
```

Figure 3.6: Introducing a TBB pipeline

definition of the class `ProcessImage` is considerably more complex than the corresponding lambda expression in Figure 4.3 since the refactoring has to generate an entire class definition; moreover, since the class definition is in a different scope from the original code, the class definition has to include code to capture free variables from the original scope. In lines 2 and 3 of Figure 4.4 fields are defined which will contain pointers to the arrays `image_names` and `output` in the original code: these are filled in by the constructor on lines 14 and 15, which is applied in line 2 of Figure 4.5, at the site of the original code. Lines 5–12 in the class definition override the function call operator `()`, allowing `parallel_for` to apply instances of `ProcessImage` like functions. Note that writing class definitions like this by hand would be quite tedious and error-prone; our experience is that automatic refactoring makes it considerably easier to write parallel code.

3.2 Parallel Pipelines

We also have a refactoring to introduce pipelines using TBB constructs, again with two variants (lambda-based and functor-based). These require a `for` loop in which the body performs a chain of computations on the elements of an array, writing the eventual results to a possibly different array. Conveniently, the sequential code in Figure 4.2 is of this form, so we can use it to demonstrate the refactorings. For the lambda-based refactoring, the refactored code is quite lengthy (25 lines), and we won't discuss it in detail here.

Part of the result of the functor-based pipeline refactoring is shown in Figure 4.6. This is the code which replaces the original loop; in addition, three new classes are generated, one for each statement in the body of the loop. These are called `Closure_1`, `Closure_2` and `Closure_3`, and their definitions (omitted to save space) are quite similar to that in Figure 4.4.

The refactoring introduces three TBB objects (`stage_1`, `stage_2` and `stage_3`) of type `filter_t`, representing the three stages of the pipeline. In line 6 these are combined into a single pipeline using the `&` operator (overloaded by TBB) and executed using TBB's `parallel_pipeline` function. The first parameter (16 here) represents the maximum number of stages of the pipeline which will be run in parallel. This value (which could be some more complex expression) is supplied by the user when the refactoring is initiated, as are the prefixes (`Closure` and `stage`) for the class definitions and stage names.

3.3 Remark on components

As can be seen from the above examples, the character of the refactored code is quite different depending on whether lambda expressions or function objects are used. Lambda expressions are quite concise and only require modification of the source code are the original site; however, the refactored code can be quite hard to read. In contrast, function objects lead to quite readable code at the expense of introducing large class definitions into the source code. There is however a possible advantage in that function objects have the potential to be re-used at multiple points. In our current TBB refactorings, component introduction happens automatically during other refactorings, but in future we may introduce a separate refactoring to introduce function objects. Our FastFlow refactorings already use this technique, so this would make the refactoring process more uniform; it would also give the user finer control over the structure of TBB refactorings.

Chapter 4

Experiments

In this section, we evaluate the effectiveness of the RPL (Section 3) and the refactoring system (Section 4) for parallelising two realistic C++ use cases – Image Convolution and Ant Colony Optimisation. For each use case, we go through the following steps: i) starting from a sequential application, we abstract the structure into an RPL expression; ii) using the RPL shell and pattern rewriting rules from Figure 3.1, we generate alternative equivalent parallel structures (parallelisations) that have the same semantics as the original application and we pick those that have the best (estimated) runtime; iii) we generate *refactoring recipes* for parallel structures from the previous step; iv) for each recipe, we apply the corresponding refactorings (described in Section 4) to the sequential application, resulting in different parallel versions of the code; and, v) we evaluate the speedups (compared to the original sequential version) of the parallel versions from the previous step. To demonstrate the generality of our approach, for each recipe in step iv) we produce FastFlow, TBB and OpenMP versions of the original application (using refactorings from Section 4). Our evaluations of the speedups were conducted on three different parallel machines, *titanic*, *xookik* and *power*. Specifications of these machines are given in Figure 5.

	titanic	***xookik***	***power***
Arch	AMD	Intel	IBM
Proc	Opteron 6176	Xeon x5675	Power8
Cores	24	6	20
Threads	24	12	160
Freq.	2.3 GHz	3.06 GHz	3.69 GHz
L2 Cache	24 x 512 Kb	12 x 256 Kb	20 x 512 Kb
L3 Cache	4 x 6 Mb	2 x 12 Mb	20 x 8 Mb
RAM	32 GB	48 GB	256GB

Figure 4.1: Experimental Setup

4.1 Image Convolution

Image convolution is a technique widely used in image processing applications for blurring, smoothing and edge detection. We consider an instance of the image convolution from video processing applications, where we are given a list of images that are first read from a file and then processed by applying a filter. Applying a filter to an image consists of computing a scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels:

$$out(i,j) = \sum_{m}\sum_{n} in(i-n, j-m) \times filt(n,m) \tag{4.1}$$

The main hotspot for parallelisation is a **for** loop that iterates over the images, reading each one from a file and applying the filter:

```
for (int i=0; i<nr_images; i++) {
  in_image[i] = read_image(i);
  out_image[i] = process_image(in_image[i]);
}
```

Using the RPL syntax, we can abstract the structure of this code as

seq (comp (seq (read_image), seq (process_image))).

In order to parallelise the code, we can convert the comp into pipe and each of the three seq into map (using, respectively, the *pipe_intro* and *map_intro* transformations from Figure 3.1). There are, therefore, 15 (2^4 minus sequential version) different parallelisations of the above code. We will focus on the three that have the best estimated performance (via RPL shell):

1. seq (pipe (seq (read_image), map (process_image)))
2. seq (pipe (map (read_image), map (process_image)))
3. map (comp (seq (read_image), seq (process_image)))

Each of these parallelisations yields a different refactoring recipe. For example, parallelisation 2) leads to the following refactorings steps:

1. C_1 = Apply *identify-task* to read_image;
2. C_2 = Apply *identify-task* to process_image;
3. P_1 = Apply *introduce-pipeline* to comp;
4. F_1 = Apply *introduce-farm* to C1;
5. F_2 = Apply *introduce-farm* to C2;

Similar recipes can be derived for other parallelisations. We can then (semi-automatically) apply these recipes in order using our refactorings to get different parallelisations of the application. Figure 5.2 shows the speedups of different parallel versions of the application using FastFlow, TBB and OpenMP on *titanic* and *xookik*. In the figure, $(m \mid s)$ denotes the parallelisation where the `comp` is refactored into a pipeline and its first stage, `seq` (read_image), is refactored into a map; $(s \mid m)$ denotes the similar parallelisation, but where the second pipeline stage, `seq` (process_image), is rewritten into a map instead of the first; and, m denotes the parallelisation where the top-level `seq` is refactored into a map. We can observe similar results on all three architectures. When a smaller number of threads is used, all parallelisations with all models perform similarly. For larger numbers of cores (e.g. 5 on *titanic*), $(s \mid m)$ parallelisation stops scaling with all models. The m and $(m \mid m)$ parallelisations continue to scale well, reaching speedups of about 20, 10 and 32 on *titanic*, *xookik* and *power*, respectively. From these experiments, we can make three conclusions. Firstly, we are able to obtain very good speedups using our approach for this example. Secondly, the performance under all three libraries (OpenMP, TBB and FastFlow) is similar on all systems with all parallelisations. Therefore, how well the solution performs depends mostly on the way in which it is parallelised. Thirdly, it is crucial to choose the appropriate parallelisation, since the best parallelisations give significantly better speedups than others.

4.2 Ant Colony

Ant Colony Optimisation (ACO) [4] is a metaheuristic used for solving NP-hard combinatorial optimisation problems. In this deliverable, we apply ACO to the Single Machine Total Weighted Tardiness Problem (SMTWTP) optimisation problem, where we are given n jobs and each job, i, is characterised by its processing time, p_i (p in the code below), deadline, d_i (d in the code below), and weight, w_i (w in the code below).The goal is to find the schedule of jobs that minimises the total weighted *tardiness*, defined as $\sum w_i \cdot \max\{0, C_i - d_i\}$ where C_i is the completion time of the job, i. The ACO solution to the SMTWTP problem consists of a number of iterations, where in each iteration each ant independently computes a schedule, and is biased by a *pheromone trail* (t in the code below). The pheromone trail is stronger along previously successful routes and is defined by a matrix τ, where $\tau[i, j]$ is the preference of assigning job j to the i-th place in the schedule. After all the ants compute their solutions, the best solution is chosen as the "running best"; the pheromone trail is updated accordingly, and the next iteration is started. The main part of the program is given below:

```
for (j=0; j<num_iter; j++) {
  for (i=0; i<num_ants; i++)
    cost[i] = solve (i,p,d,w,t);
  best_t = pick_best(&best_result);
```

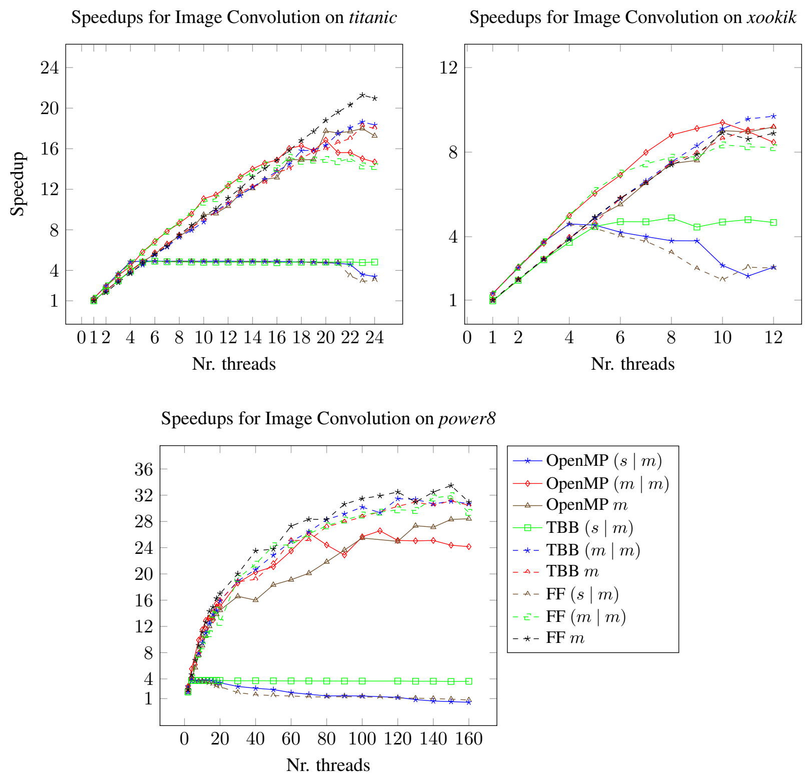


Figure 4.2: Image Convolution speedups on *titanic*, *xookik* and *power*. Here, $\mid$ is a parallel pipeline, m is a parallel map and s is a sequential stage.

```
5   for (i=0; i<n; i++)
6     t[i] = update(i, best_t, best_result);
7 }
```

Since pick_best in Line 4 cannot start until all of the ants have computed their solutions, and the **for** loop that updates t cannot start until pick_best finishes, we have implicit ordering in the code above. Therefore, the structure can be described in the RPL with:

seq (solve) ; pick_best ; seq (update)

where ; denotes the ordering between computations. Due to an ordering between solve, pick_best and update, the only way to parallelise the sequential code is to convert seq (solve) and/or seq (update) into maps. Therefore, the possible parallelisations are:

1. map (solve) ; pick_best ; update
2. solve ; pick_best ; map (update)

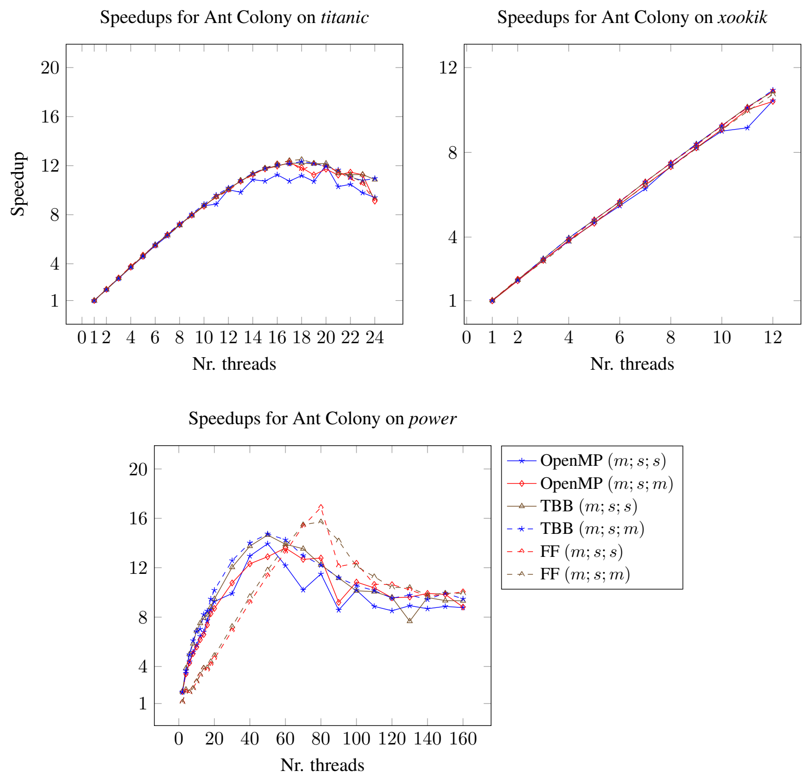


Figure 4.3: Ant Colony Optimisation Speedups on *titanic*, *xookik* and *power*. Here, ; is normal function composition, m is parallel map and s is sequential stage.

3. map (solve) ; pick_best ; map (update)

Since solve dominates the computing time, we are going to consider only parallelisations 1) and 3). Speedups for these two parallelisations, on *titanic*, *xookik* and *power*, with a varying number of CPU threads used, are given in Figure 5.3. In the figure, we denote map by m and a sequential stage by s. Therefore, $(m\ ;\ s\ ;\ s)$ indicates that solve is a parallel map, pick_best is sequential and update is also sequential. From the Figure 5.3, we can observe (similarly to the Image Convolution example) that speedups are similar for all parallel libraries. The only exception is FastFlow on *power*, which gives a slightly better speedup than the other libraries. Furthermore, both of the parallelisations give approximately the same speedups, with the $(m\ ;\ s\ ;\ m)$ parallelisation using more resources (threads) altogether. This indicates that it is not always the best idea to parallelise everything that can be parallelised. Finally, we can note that none of the libraries is able to achieve linear speedups, and on each system speedups tail off after certain number of threads is used. This is due to a fact that a lot of data is shared between threads and data-access is slower for cores that are further from the data. The maximum speedups achieved are 12, 11 and 16 on *titanic*, *xookik* and *power*.

5. Conclusion

In this deliverable, we presented a high-level domain-specific language, Refactoring Pattern Language (RPL), that can be used to concisely and efficiently capture parallel patterns, and therefore describe the parallel structure of an application. RPL can also capture extra-functional parameters of patterns, such as the service time and parallelism degree. We also demonstrated how RPL can be used in the design stage of application development, to experiment with different parallel structures of the same application to obtain the best parallelisations. We described a set of refactorings that allow semi-automatic implementations of the desired parallel structure (described in RPL) to the sequential application code. To demonstrate generality, our refactorings target three different pattern/skeleton libraries – OpenMP, Intel TBB and FastFlow. Finally, we evaluated how the RPL and refactorings can be used to parallelise two realistic C++ use cases, Image Convolution and Ant Colony Optimisation, obtaining very good speedups on three different architectures - Intel, AMD and Power. As future work, we plan to extend RPL to support additional patterns, such as Reduce, Stencil and Divide-and-Conquer. We also plan to evaluate RPL and our refactorings on larger use-cases. We also plan to extend the prediction model in the RPL shell to use a more sophisticated method based on machine learning for estimating the values of the extra-functional attributes of the patterns.

Bibliography

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating Code on Multi-cores with FastFlow. In *Proc. Euro-Par 2011*, pages 170–181, 2011.

[2] Christopher Brown, Vladimir Janjic, Kevin Hammond, Holger Schöner, Kamran Idrees, and Colin W. Glass. Agricultural Reform: More Efficient Farming Using Advanced Parallel Refactoring Tools. In *Proc. PDP 2014*, pages 36–43, 2014.

[3] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[4] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[5] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[6] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.