Project no. 644235

REPHRASE

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

Software for the Advanced Refactoring Tool
D2.6

Due date of deliverable: 24

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

Responsible institution: USTAN
Editor and editor's address: Chris Brown, USTAN

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	30/03/17	Chris Brown	USTAN	Completed first draft
2	26/04/17	Kenneth MacKenzie	USTAN	Added some technical content
3	3/5/17	Chris Brown	USTAN	Completed final version
4	23/03/18	Chris Brown	USTAN	Minor typos

Contributions Per Partner

Institution	Contribution
USTAN	Introduction and description of the refactoring tool (Sections 1, 2 and 3)
UNIPI	GrPPI interface (part of Section 2)

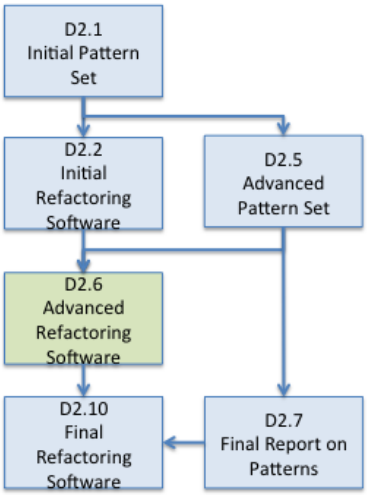

Figure 1: Dependencies between deliverables

Executive Summary

This deliverable reports on the second phase of Task 2.5 *Pattern Based Refactoring Tool Support*. In this second phase, we extend the refactoring tool to target GrPPI, the abstract pattern interface from D2.5. In particular we introduce new refactorings that target farms and pipelines in GrPPI, with refactoring extensions to introduce pipeline and farm refactorings. With this new refactoring interface, software developers can take advantage of the general abstract mechanisms of GrPPI to target different underlying pattern implementations, including PThreads, TBB, OpenMP, FastFlow and CUDA.

Contents

Chapter 1

Introduction

This deliverable presents new refactorings that target the GrPPI [2, 3] framework. GrPPI is a pattern interface library that takes advantage of C++ template metaprogramming to provide a general interface to pattern programming, hiding away the complications of the implementation mechanism. Furthermore, GrPPI also provides a *generic* interface to pattern programming in C++, building on top of many different pattern implementations, such as TBB, OpenMP, PThreads and CUDA.

The refactorings presented in this deliverable focus on the pipeline and farm patterns, which form the basis and fundamental set of patterns used in GrPPI. As more advanced patterns become available in GrPPI, supporting the pipeline and farm patterns will make it easy to extend our refactoring set to support more complex refactorings.

The refactorings are built into the RePhrase refactoring tool, known as ParaFormance (http://www.paraformance.com), and are integrated into Eclipse, as in the previous refactoring deliverable, D2.2. Furthermore, they make use of the CDT and Clang infrastructures for parsing C++ programs into intermediate forms, so that transformations and analysis can be performed on the AST.

Our refactorings make use of some user intervention, by marking pipeline stages in the code using C++ pragmas: this is to help the refactoring process by relying on some programming knowledge.

Finally, we intend for our GrPPI refactoring to be applied to the use-cases in WP6 by the use-case partners. A further WP6 deliverable will report on the refactorings and tool usage.

Chapter 2

GrPPI Refactorings: design and implementation

2.1 GrPPI

GrPPI [2, 3] is a parallel pattern interface which uses C++ template metaprogramming to provide implementations of a number of parallel patterns. The GrPPI patterns are generic over a number of different parallelism models, currently including (at least partial) support for Posix threads (PThreads), OpenMP, Intel's Thread Building Blocks (TBB), FastFlow, and CUDA Thrust. This enables users to write their application without needing to know the details of a particular model, and to easily switch between models. This strategy greatly reduces the load on the programmer.

2.1.1 Structure of GrPPI patterns

The basic GrPPI model is of a *pipeline* which processes a stream of data items: items are produced by a *source*, processed by some intermediate components, and then disposed of by a *sink*. The individual stages can all be executed in parallel, so that an item can be processed by stage number n at the same time as the following item is being processed by stage $n-1$ (and the item after that by stage $n-2$, and so on): see Figure 2.1.

There are various types of intermediate stage, some of which may themselves be carrying out parallel computations internally. Pipeline stages are represented by C++ lambda expressions, or by GrPPI constructs containing lambda expressions, and are supplied as arguments to the GrPPI `Pipeline` template.

Data sources. A source produces a stream of items of some type `T`. The output of the source is in fact of type `optional<T>` so that the pipeline can tell when

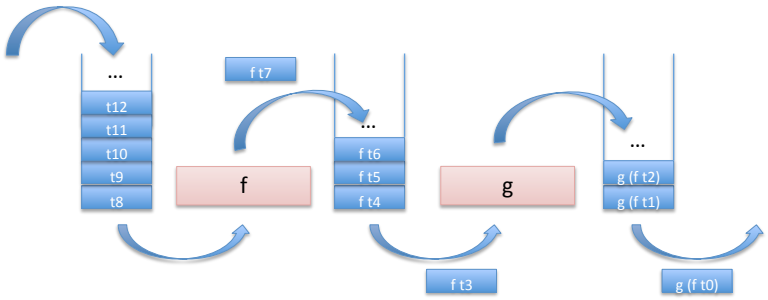

Figure 2.1: A Parallel Pipeline Pattern.

the input stream has come to and end[1].

Data sinks. A sink takes a single item of data and disposes of it in some way, returning `void`. For example, a sink might write an item to a file, or store it in an array or vector.

Intermediate stages. GrPPI provides several types of intermediate stage, including:

- Sequential stages, containing purely sequential code
- Farms, which can process multiple items in parallel
- Filters, which can discard items in the input stream based on some predicate
- Accumulators, which can amalgamate several stream items into a single item for processing by the next stage

See D2.1 and D2.4 for full details.

Pipeline stages are usually represented by C++ lambda expressions, perhaps wrapped inside GrPPI templates. A simple example is given below: this reads a number of integers from a file, squares each one, and writes the results to `std::cout`. We assume that the variable `is` is of type `istream` and is associated with a text file containing a list of integers.

```
parallel_execution_ff p{};
parallel_execution_ff f{};

Pipeline(p,
   [&]() {
       int i;
```

[1]See http://en.cppreference.com/w/cpp/utility/optional for information about the `optional` type. This is a proposed extension for C++17 which is not available in most current C++ compilers; GrPPI provides its own implementation.

```
        if (is >> i) return optional<int>(i);
        else return optional<int>();
    },

    Farm(f,
       [&](int k) {
          return k*k;
       }),

    [&](int n) {
       cout << n << endl;
    }
);
```

The two `parallel_execution_ff` declarations introduce GrPPI objects which describe the execution model to be used by the parallel pipeline components. The `ff` part indicates that the FastFlow execution model should be used; to use OpenMP (say) instead, one would replace these with `parallel_execution_omp`.

The first stage reads integers from the file and feeds them into the pipeline. The second stage squares the input items; since it is a `Farm` object, multiple items are processed in parallel and the results may not be output in the same order as the inputs. The final stage prints the squared data items to the standard output stream.

Chapter 3

The Refactoring Tool

As described in our earlier deliverable D2.2 [1], we have implemented an Eclipse plugin for performing parallelism-related refactorings. The main purpose of the tool is to allow users to semi-automatically introduce parallelism into existing sequential code while safely preserving the semantics of the original program. We have now extended the tool to include prototype refactorings using GrPPI constructs: this section will describe the design and implementation of the new refactorings.

3.1 Eclipse/CDT

Eclipse [5] is a well-known open-source Integrated Development Environment implemented in Java. The CDT project (C/C++ Development Tooling) provides a number of Eclipse plugins which convert Eclipse into a C/C++ IDE, providing features such as code completion, integrated documentation lookup, quick fixes for common programming errors, and refactoring.

We have implemented an Eclipse plugin which adds functionality to CDT which allows the user to refactor C++ code to easily introduce parallelism. Our tool also performs safety checks to ensure that the introduction of parallel constructs will not lead to semantic problems such as data races.

3.2 Refactoring in CDT

CDT parses C++ source code to produce an *Abstract Syntax Tree* (AST) containing nodes which are object from a hierarchy of Java classes representing various C++ syntactic constructs (declarations, assignment statements, arithmetic operations, function calls etc).

It is worth noting that the CDT AST is not in fact particularly abstract: since it is designed for use in an IDE, the structure of the original source code must be preserved, and distinctions which would normally be eliminated in an abstract syntax tree are retained. For example, literal integer constants can be written in

numerous ways (`42`, `0x2A`, `052` (octal), and so on). Instead of replacing these with the actual value, the CDT AST preserves the precise form of the original constant as written in the source code.

On the other hand, some information is discarded during the parsing process. The AST produced by the CDT parser does not just represent the code appearing in a source file, but the so-called *Translation Unit*, where all preprocessor directives have been resolved. This means that

- All header files in `#include` directives are bodily included, and any code which they contain is also parsed and included in the AST. Because of this, ASTs can become very large if many headers are included.

- All macro invocations are fully expanded, with the expanded text being parsed and included in the AST.

- All `#if` and `#ifdef` directives are resolved: code for true directives is included, but code for false directives is discarded. This process is based on preprocessor symbol settings in the active CDT environment.

CDT also contains a built-in refactoring framework which we have taken advantage of. This provides a uniform interface where a user can highlight some code to be refactored and select a particular refactoring. Using classes supplied by the refactoring implementer, the framework can then ask for extra information, check the safety of the proposed refactoring, and display a side-by-side preview of the result of the refactoring for confirmation by the user. Internally, refactoring is performed by generating new AST nodes and supplying them as a series of edits to be carried our by the refactoring framework.

3.3 Clang

Some of our tool implementation has also been carried out using the Clang compiler infrastructure [6]. Clang is a C/C++ compiler which is designed for IDE integration. Our tools are written in C++ using the Clang `libtooling` infrastructure, which allows users to implement stand-alone tools which use the Clang front end to parse source files and generate an AST. As with Eclipse/CDT, Clang ASTs reflect the structure of the original source very closely, and again represent translation units with all preprocessor directives fully expanded.

The advantage of using Clang is that we can produce tools which can be integrated with other multiple IDEs without wholesale re-implementation being required. Most of our safety checking code is written using Clang, and we are gradually implementing refactorings in Clang as well. However, the refactorings described below have been implemented using the Eclipse/CDT infrastructure since this is currently somewhat easier.

3.4 Refactoring and GrPPI

As mentioned above, the main aim of the tool is to help users introduce parallelism into existing sequential code. Not all of the GrPPI patterns are suitable for this methodology (e.g., filters don't correspond neatly to sequential idioms, and make it difficult to refactor), but we have identified a subset which enable easy conversion of sequential code to parallel code and implemented preliminary refactorings for these. In future work we plan to enhance our existing refactorings and extend then to make use of other GrPPI patterns.

The next section gives a brief overview of how the user interacts with the IDE to perform a GrPPI refactoring; after that we will describe the technical details.

3.4.1 User interface

How should a user of our tool interact with it? We wish to make the process as simple as possible. In our earlier implementation, we dealt with fairly simple refactorings where an entire loop could be refactored easily: the user would select the loop in the IDE's user interface and then ask to parallelise it.

When pipelines are involved, the process becomes more complicated. The tool can't automatically decide where to introduce pipeline stages and what type the stages should be, since the user may wish to choose the types of the stages for themselves (however, see D2.3 [4], which describes techniques for automated pattern discovery which address these issues).

Another complication is that it is difficult to introduce pipeline stages one by one using individual refactorings, since this will probably lead to syntactically invalid programs while the process is still incomplete; the IDE will no longer be able to parse these properly, and then it will be difficult or impossible to perform further refactorings because the AST will be incomplete.

Our approach to these problems is to have the user add annotations to the program describing where the stages should go and what their types should be: the tool then uses this information to transform the program in a single refactoring. This requires relatively little effort from the programmer, but supplies enough information for major transformations to be applied automatically.

We considered three different methods of adding annotations:

- C++ code attributes[1]
- Specially-formatted comments
- Pragmas

Attributes would have been ideal, since they are attached to AST nodes and appear in the AST. Unfortunately, it appears that attributes are intended to be used for compiler extensions, and user-defined attributes are not supported. Both Eclipse

[1] http://en.cppreference.com/w/cpp/language/attributes

and Clang allow arbitrary attributes in the source code, but ones which are unknown to the compiler are discarded with a warning, and do not appear in the AST.

This leaves us with special comments and pragmas, both of which suffer from the disadvantage that they do not form part of the AST, and must be accessed separately. In both Eclipse and Clang the process for accessing comments and relating them to AST nodes is very complex, but things are slightly easier for pragmas. Also, pragmas are more obvious in the source code. These considerations led us to use pragmas.

As an illustration of the refactoring process, consider the following very simple program:

```
// #include directives omitted

int main (int argc, char **argv) {
   float A[] = {11.1,22.2,33.3,44.4,55.5,66.6,77.7,88.8,99.9};

   for (int i=0; i<9; i++) {
      float x = A[i];
      float y = x*x;
      cout << y << endl;
   }
}
```

After annotation, this becomes

```
// #include directives omitted

int main (int argc, char **argv) {
   float A[] = {11.1,22.2,33.3,44.4,55.5,66.6,77.7,88.8,99.9};

   for (int i=0; i<9; i++) {
#pragma ppi array source
      float x = A[i];
#pragma ppi farm stage
      float y = x*x;
#pragma ppi sink
      cout << y << endl;
   }
}
```

The pragmas indicate the points at which the various stages begin, and tell the refactoring tool which kind of stage should be generated. There must be one source at the beginning and one sink at the end, with any number of intermediate stages. In general, each stage may contain multiple lines. There are currently some restrictions on the structure of the code inside the stages: we will describe these later, in §3.5.2.

One disadvantage of pragmas is that compilers usually emit a warning if they encounter an unknown pragma. With Clang and gcc, this can be avoided by using the `-Wno-unknown-pragmas` compilation option.

Having annotated the code, the user then highlights the loop in Eclipse, and selects the GrPPI pipeline refactoring option, as illustrated in Figure 3.1.

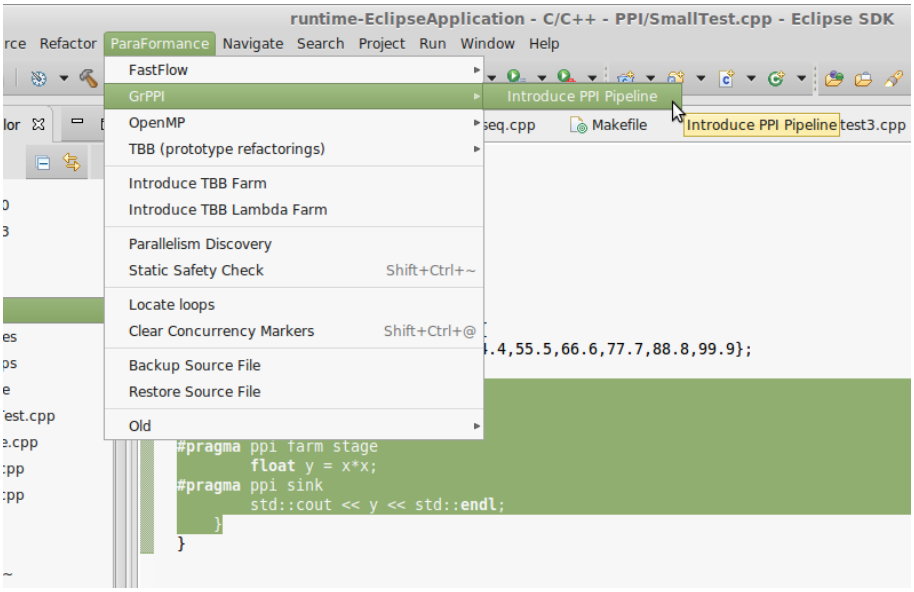

Figure 3.1: Eclipse showing the Refactoring menu and GrPPI refactorings

An input form is displayed asking the user to select the desired parallelism model (FastFlow, TBB, ...). The user is then shown a preview (and also any errors or warnings that may have occurred) and then if they choose to proceed the original program is transformed to

```
int main (int argc, char **argv) {
   float A[] = {11.1,22.2,33.3,44.4,55.5,66.6,77.7,88.8,99.9};

   parallel_execution_ff p { };
   parallel_execution_ff f { };

   Pipeline(p,
      // -------- source ----------
      [&]() {
         static int i=0;
         if (i<9) {
            auto x = optional<float>(A[i]);
            i++;
            return x;
         } else {
            return optional<float>();
         }
      },

      // -------- stage 1 ----------
      Farm(f, [&](float x) {
         float y = x*x;
         return y;
      }),

      // -------- sink ----------
      [&](float y) {
         cout << y << endl;
```

```
            return;

      });
   }
```

Any PPI header files which are required are also included automatically.

3.5 Pipelines

The basic GrPPI construct is the pipeline pattern, processing a stream of data items. Sequential C++ code uses many idioms for processing sequences of data, and we have identified a number of these which occur frequently and fit well with GrPPI's notion of streams. The next subsection describes some common idioms and how they can be expressed in terms of GrPPI constructs. For each of these idioms, there is a corresponding pragma which acts as a hint to the tool that it should look for a particular pattern in the AST and extract the information required to produce a suitable source. If the code does not conform to the expected pattern, an error message is issued.

3.5.1 Sequential C++ stream-processing idioms

C-style arrays A very common pattern, especially in older C++ code, involves the use of a for-loop to process elements of an array in sequence:

```
for (int i=0; i<n; ++i) {
   // Do something with A[i]
}
```

Here, `A` is a C-style array and `n` is its length.
The `ppi array source` pragma is used to indicate that code of this form is required, and it is fairly easy to define a GrPPI source which will produce the elements of the array in sequence:

```
[&]() {
   static int i=0;
   if (i<n) {
      auto x = optional<T> (A[i]);
      i++;
      return x;
   else return optional<T>();
}
```

Here, the elements of `A` are of type `T`: this can be discovered by examining the AST.

Note that the lambda expression contains a `static` variable `i` whose value is preserved between invocations of the lambda; GrPPI guarantees that only one copy will exist of a source object, so this strategy is safe (there will never be multiple copies each with their own private index into the array). Note also that we capture

the array size `n` by reference: this will be safe as long as we never write to `n`. We will discuss safety issues at greater length in §3.6.

If we need to write elements of an array (in a sink, for example), a similar strategy can be used.

Vectors and other containers. C-style arrays are common in existing code, but modern C++ provides numerous container types such as vectors, arrays, lists, sets and so on. These can be processed using the same idiom as for C-style arrays (using the overloaded `[]` operator), but it is more common to use iterators:

```
for(const iterator<T> i = v.begin(); i != v.end(); ++i) {
   // Do something with *i
}
```

Again, we can convert this into a GrPPI data source (using the `ppi iterator source` pragma):

```
[&]() {
   static iterator<T> i=v.begin();
   if (i != v,end()) return optional<T>(*i++);
   else return optional<T>();
}
```

Since C++11, it has also been possible to use *range-based for loops* to process elements of structures equipped with iterators:

```
for(auto x: v) {
   // Do something with x
}
```

This is entirely equivalent to the version using iterators above, and so does not present any new problems.

Files and streams. Data items from files are often processed by `while` loops. In C and older C++ this is done using file pointers and standard functions such as `fgetc`, `fgets`, `fread`, and `fscanf`, together with calls to `feof` to check for the end of the file. This is rather messy, and it's difficult to find a consistent idiom for processing data using these methods; because of this, we won't consider this style of file access any further.

In contrast, modern C++ uses much cleaner idioms using streams:

```
while (str >> x) {
   // Do something with x
}
```

Here, `str` is an object of type `ifstream` and `x` is a variable which has been declared earlier. The `>>` operator is overloaded and attempts to read an item from the file whose type matches that of the variable `x`; if it succeeds, the item read is stored in `x` and the stream is updated to point to the next item in the file; if it fails, then `str` becomes `nullptr` and the loop terminates. Exactly the same

pattern can be used with a `stringstream` object to process a stream of items (eg, characters) contained in a string.
Code conforming to this idiom is indicated by the `ppi stream source` pragma and can easily be converted to a GrPPI data source:

```
[&]() {
   T x;
   if (c >> x) return optional<T>(x);
   else return optional<T>();
}
```

Empty sources. For more complex sources which we cannot generate automatically we include a `ppi empty source` pragma which produces a lambda which just returns `optional<T>()` (where `T` is the return type of the source, supplied by the user as an argument to the pragma). This is intended to allow the user to insert suitable code themselves. The original source code between the pragma and the beginning of the next stage is included as a comment inside the lambda as a guide to the user.

3.5.2 Intermediate stages

We currently provide two types of intermediate pipeline stage: sequential stages (`ppi seq stage`) and farms (`ppi farm stage`). Both of these consist of simple sequences of statements which are wrapped inside lambda functions. In the case of a sequential stage, the lambda is inserted directly as a pipeline argument, while for a farm stage it is first wrapped in a GrPPI `Farm` construct. For farms, the refactoring also generates a declaration of an object giving the required execution model (as with `ff_parallel_execution f{}` in the example in §2.1.1). The declaration precedes the pipeline itself, and the name of the object can be supplied as an extra argument to the pragma; a name will be generated automatically if the user doesn't specify one.

Restrictions. The structure of intermediate stages is quite restricted in our current prototype. Each intermediate stage must have a single live-in variable (the output of the previous stage) and a single live-out variable which is assigned to in the final line of the stage and which will be returned by the generated lambda for input to the following stage.

In future work, we plan to relax this restriction by packaging all of the live-out variables in a stage into a C++14 `std::tuple` for forwarding to the following stage, where they will be unpacked and assigned to variables. This strategy is in part dictated by the fact that GrPPI components all require single inputs (and, like all C++ functions, can only return single results). Ostensibly, the implementation should be fairly straightforward; however, some dataflow analysis is required and this is quite complicated because of the CDT AST's concreteness, where no details are abstracted away.

Another issue concerns arrays. Within a loop, one will generally refer directly to elements `A[i]`, where `i` is the loop variable. This would be dangerous inside a pipeline, since different stages would require different values of `i`. To overcome this, we require that array elements are assigned to variables which are then used instead of direct accesses to `A[i]`. If the original code is written in this form, then the refactoring should be successful; otherwise, the variable `i` will be out of scope anyway, so the refactored code will not compile. This problem could be dealt with by a simple preliminary refactoring, but we have not yet implemented this. Similar remarks apply to container elements referenced by `*i` for an iterator `i`.

3.5.3 Sinks

Sinks are generally similar to sources, except they are simpler because no `optional` types are required. Special treatment is needed for array sinks of the form `B[i] = ...`, since an index variable must be included in the corresponding lambda. This is indicated by the `ppi array sink` pragma. Most other types of sink can be represented by simple lambdas, since they will generally consist of expressions such as `cout << x` or `v.push_back(x)`; such situations are indicated by `ppi sink`.

As with sources, we provide a `ppi empty sink` pragma which generates a lambda which merely returns without doing anything with its argument. This is intended to allow the user to insert suitable code themselves if the source is too complex. The body of the lambda includes the original source code as a comment.

3.6 Safety

There are some issues related to safety which we have not addressed yet. In particular, it is essential to ensure that no data-races occur. This means that **stages must not write into non-local data**. For example,

- No code within a pipeline must write to a variable defined outside the pipeline, although it is safe to read such variables if there are no write accesses.
- One should not call effectful functions from code which might be run in parallel with other code calling the same function.
- Different pipeline stages (or multiple copies of the same stage running in parallel) must not write to a data structure if there is any danger of two threads writing to the same element. This precludes the use of methods such as `vector::push_back` which can be non-thread-safe in certain circumstances. Note that it is safe to call such methods in sinks, which are guaranteed to run single-threaded.

Some of these conditions will be checked under T3.2 *Detection of Catastrophic Failures including Race Conditions and Deadlocks* task from WP3. In the mean-

time, the plugin does check for some of the simpler issues, and others can be precluded by the judicious use of `const` qualifiers; we hope to improve our refactorings to make use of these later.

Chapter 4

Conclusion

In this deliverable, we presented a set of new refactorings that target GrPPI: a parallel pattern interface that uses C++ template metaprogramming to provide implementations of parallel patterns. The advantage of targeting GrPPI is that it can be used as a generic pattern interface to support a number of different parallelism models, such as Posix Threads, OpenMP, TBB, FastFlow and CUDA. This simplifies the development model and also the refactoring implementation. In this deliverable we showed new refactoring that introduce GrPPI refactorings including GrPPI pipelines and Farms. This enables the refactoring tool to support heterogeneous programming through the GrPPI support for CUDA.

In the future, we will extend our refactorings to include patterns from the advanced pattern set. As more patterns become available in GrPPI, it will be fairly straightforward to extend our basic refactoring set to include them.

Bibliography

[1] Christopher Brown. Software for the initial refactoring tool. Technical report, The University of St Andrews, School of Computer Science, 04 2016.

[2] Marco Danelutto. Software for implementations of initial patterns. RePhrase Deliverable 2.4.

[3] David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sanchez, Javier García Blas, and J. Daniel García. *A C++ Generic Parallel Pattern Interface for Stream Processing*, pages 74–87. Springer International Publishing, Cham, 2016.

[4] Manuel Dolz. Report on program shaping and pattern discovery for the initial pattern set. Technical report, The University of Carlos III Madrid, 2016.

[5] Eclipse Foundation. Eclipse - an Open Development Platform. `http://www.eclipse.org`, 2009.

[6] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.