Project no. 644235

REPHRASE

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

Software for the final refactoring tool
D2.10

Due date of deliverable: M33

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

Responsible institution: USTAN
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	18/12/17	Kenneth MacKenzie	USTAN	Initial version
2	20/12/17	Adam Barwell	USTAN	Added initial examples and a brief overview of mapreduce, stencil, and divide and conquer refactorings.
3	10/1/18	Chris Brown	USTAN	General edits. Made changes to conclusions.
4	23/03/18	Chris Brown	USTAN	editorial fixes due to reviewing

Contributions Per Partner

Institution	Contribution
USTAN	Complete deliverable and source code implementation

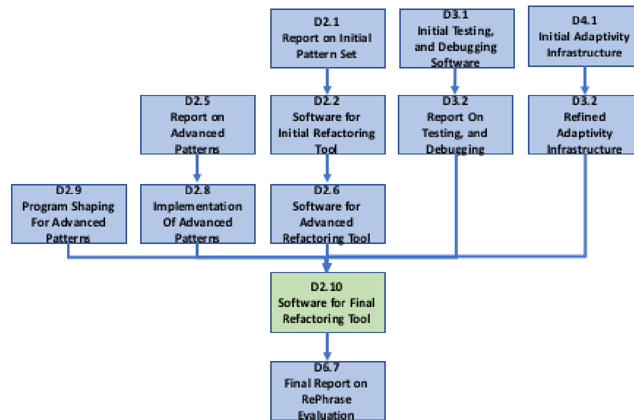

Figure 1: Dependencies between deliverables

Executive Summary

This deliverable reports on the third and final phase of Task 2.5 *Pattern Based Refactoring Tool Support*. In this final phase, we further extend the GrPPI refactorings from the previous deliverable, D2.6, to target some of the advanced pattern set. In particular, we introduce new refactorings that introduce advanced GrPPI patterns including *stencil*, *divide and conquer* and *map reduce*. In addition to these advanced patterns, we have also extended and improved the original *pipeline* refactorings as stated in D2.6.

Contents

Executive Summary 3

1 Introduction 6

2 A Background to GrPPI 7
2.1 Parallelism models 7
2.2 Compiler flags 7
2.3 Compiler versions 8

3 Refactoring 9
3.1 Refactoring interface 9
3.2 Refactoring strategy 10
3.3 Safety checking 11
3.4 While loops 12
3.5 Restrictions 13

4 MapReduce Refactoring 14
4.1 Refactoring Strategy 14
4.2 Safety Checking 16
4.3 Restrictions 17

5 Stencil Refactoring 18
5.1 Refactoring Strategy 18
5.2 Safety Checking 21
5.3 Restrictions 21

6 Divide and Conquer 22
6.1 Refactoring Strategy 22
6.2 Safety Checking 25
6.3 Restrictions 25

7 Caveats 27
7.1 General 27
7.2 Spurious template errors in CDT 27
7.3 Incorrectly rendered types 28

8 Conclusion **29**

A Pipeline example **30**

Chapter 1

Introduction

In RePhrase deliverable D2.1 we described a preliminary implementation of a pipeline refactoring targeting the GrPPI framework (`https://github.com/arcosuc3m/grppi-materials`) for parallelism in C++. We have now improved and extended this refactoring to handle much more general code, including experimental support for `while` loops. In addition, GrPPI itself has evolved to include special constructs for commonly-occurring patterns of parallelism. We have implemented experimental refactorings targeting some of these, including map-reduce, divide-and-conquer, and stencil computations. For map-reduce and stencil patterns, the refactorings support the transformation of `for` loops into calls to the relevant skeletons. The divide-and-conquer refactoring introduces the skeleton by transforming recursive functions. All three refactorings use pragma annotations to derive relevant stages of the skeleton; these pragmas must be introduced prior to applying the refactoring, and could potentially be introduced automatically.

In the rest of this document we will describe our current implementation, mention some limitations which we have encountered, and give some simple examples. As in D2.1, our refactorings are implemented as part of the ParaFormance plugin for CDT[1], an Eclipse-based C/C++ development environment.

[1] See `https://www.eclipse.org/cdt/`

Chapter 2

A Background to GrPPI

Recall that GrPPI provides a *generic* set of parallel patterns which can be used to construct parallel programs; it supports a number of parallelism models and allows the model to be changed with minimal alteration of program sources. GrPPI components can be used as building blocks to create parallel programs from scratch, but our plugin allows sequential programs to be semi-automatically *transformed* into parallel programs by recognising certain code idioms and converting them into GrPPI constructs.

2.1 Parallelism models

The current version of GrPPI supports three models of parallel execution:

- Native (pthreads)
- OpenMP
- Intel TBB

The public version of GrPPI does not currently support FastFlow, but the plugin does allow one to use FastFlow features in case GrPPI support becomes available later.

2.2 Compiler flags

Certain compiler flags are required for each of these models:

- Native: `-pthread`
- OpenMP: `-fopenmp -DGRPPI_OMP`
- TBB: `-DGRPPI_TBB -ltbb`

- FastFlow: `-DGRPPI_FF`

Appropriate `-I` options may also be required to access relevant header file directories, including the GrPPI headers.

If the user wishes to compile an application with different parallel execution models within Eclipse then it may be necessary to create corresponding Eclipse build configurations using the appropriate flags (and also perhaps corresponding makefiles, depending on how Eclipse manages the build process).

GrPPI uses the `optional` type. This is fully supported in C++17 and is available as `std::experimental::optional` with C++14. Accordingly, programs using GrPPI must be compiled with at least `-std=c++14` or (equivalently) `-std=c++1y`.

2.3 Compiler versions

We have tested our refactorings with g++ version 7.1.1, and the GrPPI developers report that it works with version 6.2. Earlier g++ versions may not work.

GrPPI currently **does not work** with any version of Clang, up to and including version 6.0.0.

There are also some difficulties with the Eclipse/CDT interface, which reports spurious template errors when some of the GrPPI constructs are used: see the "Caveats" section later for more about this. Despite the reported errors, the code can still be built and run.

Chapter 3

Refactoring

The ParaFormance Eclipse plugin supplies a refactoring which enables semi-automated introduction of GrPPI constructs to parallelise sequential C++ code. This is described in RePhrase deliverable D2.6, but the ParaFormance interface has changed slightly to conform to recent changes in GrPPI. We have also eliminated many of the restrictions in D2.6, allowing the tool to handle much more general code, and we have simplified the interface by removing the necessity to specify the type of object which is acting as a source for items in a GrPPI pipeline.

3.1 Refactoring interface

The plugin provides a refactoring which allows the user to convert a C++ `for` loop into a GrPPI pipeline containing one or more stages which are executed concurrently to process a sequence of data items indexed by the loops. We currently support sequential stages which process a single data item at a time and farm stages which process multiple items in parallel. Note that the refactoring imposes the mild restriction that the body of the loop must be a compound statement enclosed in braces: `{...}`.

Prior to refactoring, the user must manually insert pragmas into the body of the loop indicating the pipeline stages (eventually, a suitable analysis might allow this process to be automated). The currently available pragmas are

- `#pragma grppi seq stage`
- `#pragma grppi farm stage`
- `#pragma grppi farm stage [number of threads]`

A GrPPI farm stage must specify a number of threads to execute the farm. By default this is 4, but the pragma can give an alternative value (`number of threads` above) which will be used in the refactored code. This will usually be a literal integer or a variable name, but in fact any trailing text in a `grppi farm`

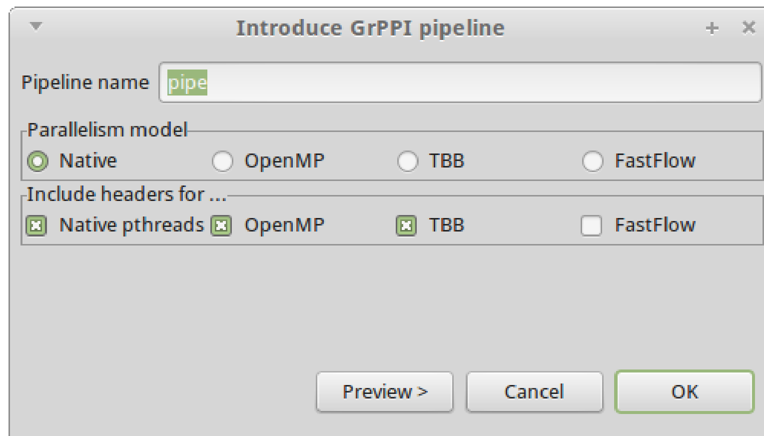

Figure 3.1: Input Form

`stage` pragma will be inserted directly into the refactored code. There is no check of syntactic validity, so it is up to the user to write something sensible.

Note that it is possible to create a pipeline which consists of a single farm stage: this is equivalent to running multiple (and possibly all) iterations of the loop concurrently.

Once the pragmas have been inserted, the user highlights the relevant loop and selects `GrPPI > Introduce GrPPI Pipeline` in the `ParaFormance` menu. A form appears allowing the user to enter a name for the pipeline and select the parallelism model (see Figure 1).

The intention here is that the user selects an initial parallelism model but is able to select extra headers to allow the parallelism model to be changed later. For example, one could change `parallel_execution_native` to `parallel_execution_tbb` in the program source; if the TBB headers have already been included then no other changes will be required in the source. These settings will be remembered between invocations of the refactoring, but will be reset to a default if Eclipse is restarted. The form also allows the user to enter a name for the pipeline: the default this is `pipe` (possibly with a numeric suffix to avoid conflicts with other variables which are in scope), but any other name can be used.

Once the form has been completed, the user can select either `Preview` to display the changes which will be made, or `OK` to perform the changes directly, without a preview. They can also press `Cancel` to abandon the process and leave the program source unaltered.

Note that for Eclipse to parse and build the refactored code, appropriate include paths and compiler options may need to be added to the project build configuration.

3.2 Refactoring strategy

To refactor a `for` loop, the loop must be of one of the forms below:

- `for (T x=e1; e2; e3) { ... }`
- `for (x=e1; e2; e3) { ... }`

Here `x` is a variable name and `T` is a type. As usual, `e1` is an initial value for `x`, `e2` is some limiting condition, and `e3` is an expression which updates the value of `x`; `e2` may be empty, in which case the pipeline will run forever, but `e3` must not be empty. The crucial restriction here is that the loop must declare or initialise a single variable in the initialiser; this variable then represents data items passing through the pipeline.

These patterns allow us to refactor a number of common types of loop: for example `for (int i=0; i<N; ++i) ...` and `for(auto i = v.begin(); i != v.end; ++i) ...`.

The refactoring procedure creates a GrPPI `pipeline` object and inserts a source object which returns consecutive values for `x` in the form of `optional` items, with an empty value when there are no data items left.

The pipeline stages in the loop body are converted into a sequence of lambda expressions following the source. If a variable is declared inside a loop stage but required in a later stage it will be returned from the corresponding lambda and passed as an argument to the following lambda: if there is more than one such variable then they will be returned packed into a tuple which will then be unpacked into variables in the next stage.

Variables which are declared outside the loop are captured in the lambda expressions *by reference*, which means that the lambdas can modify them: this may lead to race conditions.

3.3 Safety checking

The refactoring carries out some very basic safety checks. If the loop body contains a `return`, `break`, `continue`, `goto`, or `exit` a fatal error will be issued, since these statements will either fail to make sense inside the refactored code, or could cause an iteration of the loop to terminate when later iterations have already begun to execute, almost certainly violating the intended semantics of the loop.

The plugin also checks for write accesses to variables declared outside the loop and will warn of a potential race condition if it finds any (because multiple threads may attempt to read and write the variable simultaneously). However it will not warn of possible race conditions on array elements or object fields, and makes no attempt to track aliasing which might allow simultaneous access to a variable under different names, or via different pointers. The ParaPhrasing menu does contain an entry called `Concurrency Safety Check` which invokes an external tool to check for more complex concurrency problems (and in some cases automatically repair them). However, this is designed for parallelisation of single loops, and it may not detect pipeline-specific problems; moreover it is still undergoing development and may allow some potential race conditions to escape unnoticed.

Other problems may arise from the fact that the loop body is broken up into several parts with data being passed between them. For example:

- If a large data structure is allocated on the stack (more precisely, with automatic storage duration) in one stage and used again in a later stage, it will be *copied* as a parameter, which could be problematic if the structure is large.
- If a pointer is created to a stack-allocated variable in one stage and then used in a later stage, the data to which it is pointing may have changed, or the memory may be in use for something completely unrelated.

Ultimately the user must be careful not to introduce race conditions or other concurrency problems. Work is ongoing to detect these problems automatically, but is not yet complete.

3.4 While loops

The plugin also has some experimental support for refactoring `while` loops. Given a general `while` loop it can be difficult to decide whether it processes a sequence of items in a manner suitable for pipelining: for example, the input to one iteration may be the output of the previous iteration (as would happen in many types of simulation, where each iteration of a loop represents the evolution of some system over a single timestep), and this dependence makes the loop unsuitable for pipelining. However, we have identified a number of commonly-occurring patterns which are plausible candidates for pipelining and which the tool will refactor.

1. `while (x=e) { ... }`
2. `while ((x=e1) op e2) { ... }`
3. `while (obj >> x) { ... }`

In case 1, we assume that `x` is being given a new value by some expression `e` which presumably performs some side-effect; for example `while (x=*p++) ...`.

In case 2, we again assume that `x` is being supplied with new values by `e1` and the value is being compared with e2 using some binary operator `op`; for example `while ((x=getc(stream)) >= 0) ...`

In case 3, we assume that the >> operator is supplying new values from a stream or some other type of object. This allows us to handle common idioms such as `while (std::cin >> x) ...`.

As before, suitable pragmas should be inserted in the loop body and then `Introduce GrPPI Pipeline` selected.

We emphasise that this feature is experimental and should be used with care: it is quite possible that problems may arise which we have not anticipated. Moreover,

the plugin is not performing full dependency analysis (although reports of race conditions on the loop variable are a clear danger signal), so it is possible that the refactoring may not preserve the semantics of the loop. We hope to introduce more comprehensive safety checks in future versions of the tool.

3.5 Restrictions

GrPPI provides other types of pipeline stage, notably *filters* which allow items to be discarded from a pipeline according to some predicate, and *reductions* which allow pipeline items to be aggregated into a cumulative "total" of some type. It is somewhat difficult to recognise these automatically as part of the refactoring process (as opposed to the programmer *constructing* a parallel program by building individual components and then assembling them together); we believe that static analysis techniques could be developed to recognise patterns corresponding to filters and reductions in serial code (at least in some cases) but the tool does not at present provide any support for this.

Chapter 4

MapReduce Refactoring

In addition to Pipeline and Farm skeletons, GrPPI provides more advanced skeleton interfaces. These include: MapReduce, Stencil, and Divide and Conquer. In this chapter we describe a refactoring that introduces the MapReduce skeleton, and in chapters 5 and 6, we describe refactorings that introduce Stencil and Divide and Conquer patterns.

4.1 Refactoring Strategy

A `for` loop may be refactored into a MapReduce skeleton. As before, and to refactor a `for` loop, the loop must be in one of the forms below.

- `for (T x=e1; e2; e3) { ... }`
- `for (x=e1; e2; e3) { ... }`

Here, the loop must iterate over a `vector` data structure, where `T` is an `iterator` on that same vector, and where that iterator is incremented each iteration. Both an entire vector and a specific subvector can be iterated over. The refactoring derives the input vector and the range to be permuted from the initialisation and condition statements in the `for` loop. Consider the following loop example:

```
for (vector<string>::iterator it = begin(v); it != end
    (cont.)(v); it++) {
  ...
}
```

Here, the refactoring will lift the declaration of `it` in the initialiser statement, and use `it` as the start of the range in the MapReduce skeleton. Similarly, the refactoring will pass `end(v)` to the skeleton, to denote the end of the range.

In order to derive the remaining arguments to the MapReduce skeleton we rely on the introduction of pragmas. The pragmas are expected to be in the following forms:

- `#pragma grppi mapreduce init` z
- `#pragma grppi mapreduce map arg` arg `rtn` q
- `#pragma grppi mapreduce reduce arg` $arglist$ `rtn` r

The first denotes the initial value, z, of the reduction stage. This is passed directly to the MapReduce skeleton. For example, an initial value of 0.0 may be used for an operation that sums the elements of a vector. This initial value could be derived automatically by discovering the initial value of the variable, r, used to store the result of the operation.

The second pragma denotes both the argument, arg, for the map lambda argument and the returned variable, q. The lambda argument, arg, comprises both a variable and its type; e.g. `string s`. The lambda may have only one argument, therefore only one type-variable pair should be provided. In addition to denoting the arguments required to construct the map lambda, the pragma indicates the start of a sequence of statements in the body of the loop that will comprise the body of the constructed lambda.

The final pragma denotes both the arguments, $arglist$ to the reduce lambda, and the return variable, r. The lambda arguments $arglist$ comprise two type-variable pairs, and represent the two values that are to be reduced. The return variable represents both the variable returned in the reduce argument, but also the variable to which the result of the MapReduce Skeleton is assigned. Additionally, this pragma denotes the end of the list of statements that comprise the body of the map lambda, and similarly it denotes the beginning of the list of statements that will comprise the body of the constructed reduce lambda.

Consider the below example `for` loop that takes a vector of `double` values represented as `string` objects, and sums their values.

```
for (vector<string>::iterator it = begin(v); it != end
   (cont.)(v); it++) {
#pragma grppi mapreduce init 0.0
  string s = *it;
#pragma grppi mapreduce map arg string s rtn d
  double d = stod(s);
#pragma grppi mapreduce reduce arg double d double res
   (cont.) rtn res
  res = d + res;
}
```

Here, the assignment to `it` is lifted out into a statement that occurs before the loop, and *it* is passed to the skeleton as the starting element in the vector. Similarly, `end(v)` is passed to the skeleton as the final element in the vector. The first pragma means that 0.0 is passed to the skeleton as the initial value. The second pragma enables the construction of the map lambda:

```
[&](string s) {
  double d = stod(s);
  return d;
}
```

Similarly, the third pragma enables the construction of the reduce lambda:

```
[&](double d, double res) {
  res = d + res;
  return res;
}
```

In this example, the statement `string s = *it;` is removed entirely since it does not appear after the map or reduce pragmas. The statement is not needed, since it serves to convert the current iterator to value of the element it points to, which can then be used by the assignment statement that forms the map lambda body. It is not necessary to include this conversion, since the skeleton passes an element to the map lambda automatically.

Once all skeleton arguments are derived, the original loop may be entirely replaced with an assignment statement that includes a call to the MapReduce skeleton. For our above example, the result is as follows:

```
double res = map_reduce(exec,
  begin(v), end(v),
  0.0,
  [&](string s) {
    double d = stod(s);
    return d;
  },
  [&](double d, double res) {
    res = d + res;
    return res;
  }
);
```

Here, as in Chapter 2, the refactoring additionally introduces an execution policy declaration, defining `exec`, that is placed before the call to MapReduce.

4.2 Safety Checking

The same basic safety checks that are performed by the Introduce Pipe refactoring should also apply to this refactoring. In addition, the refactoring should ensure that neither the body of the map nor reduce lambdas should contain the iterator variable

as a subexpression. The refactoring should also check to ensure that no statement to be included in the map lambda depends upon any statement occurring in the body of the reduce lambda.

4.3 Restrictions

In its present form, the refactoring requires significant programmer input in the form of pragmas; these pragmas could be simplified and potentially introduced automatically. Liveness analysis could indicate which variables should be used as arguments and return values in constructed lambdas, as demonstrated in the Introduce Pipeline refactoring. Similarly, loop body statements must be ordered such that all those that will be included in the body of the map lambda must occur *prior* to those statements to be lifted into the reduce lambda. This restriction could be avoided or enforced by either preliminary refactoring to reorder statements, or by limiting pragmas to refer to a single statement in the loop body.

Chapter 5

Stencil Refactoring

This refactoring is designed to refactor two `for` loops into a call to the Stencil skeleton provided by GrPPI.

5.1 Refactoring Strategy

Unlike Introduce Pipeline and Introduce MapReduce refactorings, the introduce a Stencil Refactoring requires that two `for` loops are selected and annotated using pragmas. Two loops are required since the GrPPI Stencil skeleton has two stages: neighbourhood selection, and cell transformation.

The first loop pertains to the neighbourhood selection stage and is denoted by a pragma prior to the loop:

- `#pragma grppi stencil neighbourhood` n r

The pragma should also contain the variable, n, denoting the current neighbourhood vector, and the variable, r, that is returned in the neighbourhood lambda. The loop should be in the same form as required by the MapReduce Refactoring; e.g.

```
for (vector<double>::iterator it = begin(v); it != end
   (cont.)(v); it++) { ... }
```

In order to construct the neighbourhood stage lambda, the refactoring requires the range of operation over n. This is again derived using the initialisation and condition statements in the `for` loop. The neighbourhood lambda receives `it` and returns the variable declared in the pragma. Consider the following example that selects the elements immediately before and after the current element:

```
#pragma grppi stencil neighbourhood n r
for (vector<double>::iterator it = begin(v); it != end
   (cont.)(v); it++) {
  vector<double> r;
```

```
  if (it!=begin(v)) r.push_back(*prev(it));
  if (distance(it,end(v))>1) r.push_back(*next(it));
  n.push_back(r);
}
```

Here, `it` becomes the starting element and argument to the neighbourhood lambda; and `end(v)` becomes the final element. Since `n` is given as the neighbourhood vector, the refactoring lifts the body of the loop into the lambda without the final body statement, which appends `r` to `n`. This is instead replaced by a return statement, returning `r`, as indicated by the pragma. This produces the lambda:

```
[&](auto it) {
  vector<double> r;
  if (it!=begin(v)) r.push_back(*prev(it));
  if (distance(it,end(v))>1) r.push_back(*next(it));
  return r;
}
```

The other `for` loop is refactored into the transformer lambda, and is denoted by the pragma:

- `#pragma grppi stencil transformer` n r w

The transformer loop is expected to iterate over the vector, and using a nested loop, the neighbourhood for each particular cell. A separate vector, w, is expected to be written to in order to store the result of the Stencil transform operation. w should have the same structure as the traversed vector, and individual elements should be assigned in each iteration of the loop. This assignment is replaced with a return statement in the transform lambda. Consider again our example that finds the previous and next element of a cell; let us assume that we wish to sum the contents of these three cells, and that our transform loop has the form:

```
#pragma grppi stencil transformer n r w
for (int i = 0; i < v.size(); i++) {
  double r = v[i];
  for (vector<double>::iterator nit = begin(n[i]); nit
     (cont.) != end(n[i]); nit++) {
  r = r + *nit;
  }
  w[i] = r;
}
```

This nested loop can be refactored into the lambda:

```
[&](auto it, auto n) {
```

```
  double r = *it;
  for (vector<double>::iterator nit = begin(n); nit !=
     (cont.) end(n); nit++) {
    r = r + *nit;
  }
  return r;
}
```

Here, the body of the loop has been lifted into the body of the lambda, with the `w` assignment transformed into a return statement. The lambda takes two arguments: `it`, an iterator pointing to the current element; and `n`, the vector of neighbours that is produced by the neighbourhood lambda for the element referenced by `it`.

Having derived both stages of the Stencil skeleton, the refactoring replaces the two original loops with a single call to the skeleton. The skeleton takes six arguments:

1. the execution policy for the skeleton, selected by the programmer;
2. an iterator indicating the beginning of the vector to which the Stencil is to be applied, derived from the neighbourhood loop initialisation statement;
3. an iterator indicating the end of the vector to which the Stencil is to be applied, derived from the neighbourhood loop condition;
4. an iterator indicating the beginning of the vector to which the Stencil writes the output of the transformation operation, derived from the transformation pragma and the neighbourhood loop initialisation statement;
5. the transformation lambda; and
6. the neighbourhood lambda.

In our summation example, the refactoring would produce:

```
stencil(exec, begin(v), end(v), begin(w),
  [&](auto it, auto n) {
    double r = *it;
    for (vector<double>::iterator nit = begin(n); nit
       (cont.)!= end(n); nit++) {
      r = r + *nit;
    }
    return r;
  },
  [&](auto it) {
    vector<double> r;
    if (it!=begin(v)) r.push_back(*prev(it));
```

```
    if (distance(it,end(v))>1) r.push_back(*next(it));
    return r;
  });
```

5.2 Safety Checking

The same basic safety checks that are performed by the Introduce Pipeline refactoring should also apply to this refactoring. In addition, the refactoring expects that in the neighbourhood loop, only the final statement in the loop body can refer to the neighbourhood vector, n, and that statement should be an append operation. Similarly, in the transformer loop, the nested loop should access only the neighbourhood for that cell; and the final statement in the outer loop body should be an assignment statement, assigning to the cell in the output vector, w, that corresponds to the cell in the vector being iterated over. No other reference to either n or w should occur in their respective loop bodies.

5.3 Restrictions

Similar to the Introduce MapReduce refactoring, this refactoring requires the code that it transforms to be in a particular form. To introduce a Stencil, two annotated `for` loops are required. In principle, it should be possible to derive a means to automatically detect the statements and expressions that constitute neighbourhood generation and transformation, however this is ongoing and future work. Should this be implemented, pragmas could be used to label individual statements or expressions, as in the Introduce MapReduce refactoring. Another restriction shared between Introduce Stencil and Introduce MapReduce refactorings is the requirement that the pragmas indicate the variables to be used as part of the Stencil. Once again, these could, in principle, be derived automatically.

Chapter 6

Divide and Conquer

This refactoring is designed to refactor a recursive function declaration into a call to the Divide and Conquer skeleton provided by GrPPI.

6.1 Refactoring Strategy

This refactoring transforms a recursive function that performs a Divide and Conquer operation on a vector. As before, the operation is called on a range of a vector denoted by a `pair` of iterators. The function to be refactored should take the iterator pair as its sole argument. Pragmas are used to annotate the function body, denoting divider, solver, and combiner parts:

- `#pragma grppi dc divider start` ps
- `#pragma grppi dc divider end`
- `#pragma grppi dc solver` p s
- `#pragma grppi dc combiner` $s1$ $s2$ r

The first and second pragmas are used to denote the range of statements that will be lifted into the divider lambda. The third pragma denotes the `for` loop that will be refactored into the solver lambda. The final pragma denotes the beginning of the statements to be lifted into the combiner lambda; the range of statements stops at the end of a body of statements. The first, third, and final pragmas are additionally annotated with variables returned by their respective lambdas. The first pragma indicates the intermediate vector of pairs, ps, that will be returned as a result of the divider lambda. The third pragma is annotated with the solution, s, of an operation applied to each divided element in ps that will be returned as a result of the solver lambda. The final pragma is annotated with the solutions, $s1$ and $s2$, that are to be combined, as well as the result of the combination, r.

The function is expected to begin with a divider section of statements, which produces the vector ps, containing either one or more elements. The combiner

section of statements is expected to use $s1$ and $s2$ as part of an operation; the result, r, will be returned by the combiner lambda. The solver section is not required, but should comprise a `for` loop that applies some operation to each element in ps. The loop body will comprise the body of the solver lambda, which takes p as its argument and returns s.

To illustrate this, we consider a merge sort example that sorts a vector in-situ and where the main body of the sorting algorithm is defined:

```
void msBody(pair<vector<int>::iterator,vector<int>::
   (cont.)iterator> pair) {
#pragma grppi dc divider start pairs
  vector<int>::iterator first = pair.first;
  vector<int>::iterator last = pair.second;
  vector<pair<auto,auto>> pairs;
  if (last - first > 1) {
    vector<int>::iterator middle = first + (last -
       (cont.)first) / 2;
    pair<auto,auto> pair1 = make_pair(first, middle);
    pair<auto,auto> pair2 = make_pair(middle, last);
    pairs = {pair1,pair2};
  } else {
    pairs = {pair};
  }

#pragma grppi dc divider end
  if (pairs.size == 2) {
    msBody(pairs[0]);
    msBody(pairs[1]);

#pragma grppi dc solver pairs
    for (vector<pair<auto,auto>>::iterator p = begin(
       (cont.)pairs);
            p != end(pairs); p++) {
      *p = id(*p);
    }

#pragma grppi dc combiner pairs[0] pairs[1] r
    std::inplace_merge(pairs[0].first, pairs[0].second
       (cont.), pairs[1].second);
    pair<auto,auto> r = make_pair(pairs[0].first,
       (cont.)pairs[1].second);
  }
}
```

Here, `pair` is a pair of iterators, `first` and `last`, indicating the subvector inspected by this recursive call. In the divider section, and when the subvector has at least 2 elements, the midpoint of the subvector is calculated and stored as an iterator, `middle`. Two pairs are created, `pair1` and `pair2`, and stored in the intermediate pair vector, `pairs`, to be used as the arguments to two recursive calls. When the subvector contains a single element, `pairs`, contains `pair`. The lambda is constructed using the statements in between the vector, as well as the argument to the function, `pair`:

```
[&](pair<auto,auto> pair) -> vector<auto> {
  vector<int>::iterator first = pair.first;
  vector<int>::iterator last = pair.second;
  vector<pair<auto,auto>> pairs;
  if (last - first > 1 ) {
    vector<int>::iterator middle =
      range.first + (range.second - range.first) / 2;
    pair<auto,auto> pair1 = make_pair(first, middle);
    pair<auto,auto> pair2 = make_pair(middle, last);
    pairs = {pair1,pair2};
  } else {
    pairs = {pair};
  }

  return pairs;
}
```

Once divided, and when `pairs` contains two subvectors, `msBody` is recursively applied to those subvectors. In this example, the solver section is redundant, only applying the identity function to each pair, but included to illustrate the refactoring. The solver lambda is derived by taking the body of the `for` loop, and the initialiser statement as the lambda argument. The return statement is constructed using the right-hand side of the assignment to `*p`.

```
[&](pair<auto> p) {
  return id(*p);
}
```

The combiner lambda is derived by taking the statement immediately below the pragma, replacing `pairs[0]` and `pairs[1]` for valid variable names, `r1` and `r2`, that are then used as the arguments to the constructed lambda. The returned variable is `r`, effectively recreating the original `pair` input of the original function.

```
[&](pair<auto,auto> r1, pair<auto,auto> r2) {
  std::inplace_merge(r1.first, r1.second, r2.second);
  pair<auto,auto> r = make_pair(r1.first, r2.second);
```

```
  return r;
}
```

The derivation of all three lambdas enables the refactoring to replace the body of the function with a call to the skeleton.

```
void msBody(pair<vector<int>::iterator,vector<int>::
   (cont.)iterator> pair) {
  parallel_execution_native exec { };

  divide_conquer(exec, pair
    [&](pair<auto,auto> pair) -> vector<auto> { ... },
    [&](pair<auto,auto>) { ... },
    [&](pair<auto,auto> r1, pair<auto,auto> r2)
      { ... }
  );
}
```

Here, we have omitted the bodies of each lambda for clarity. As before, `exec` is the execution policy, selected by the programmer and introduced by the refactoring. Should the solver loop not be present, a placeholder lambda that immediately returns its argument is introduced.

6.2 Safety Checking

The same basic safety checks that are performed by the Introduce Pipeline refactoring should also apply to this refactoring. In addition, the solver and combiner sections should not depend upon the original function argument since they cannot access it as a lambda in the c++. The solver loop may only reassign the current iterator value once; more than once may indicate dependencies that could cause problems. Similarly, the solver loop may only apply its operation to each element in ps independently of other elements.

6.3 Restrictions

This refactoring is primarily limited by the form of function that it can transform. Not only must the function be recursive, but the function body must be divided into clearly identified sections. Additionally, the pragmas identifying these sections require additional variable names in order to facilitate the construction of the respective lambda. As in the other refactorings described in this document, the automatic identification of these sections is currently ongoing and future work. The refactoring may also be extended to facilitate the transformation of functions that are valid Divide and Conquer operations but are not expressed as recursive

functions. Any two equivalent Divide and Conquer operations described, e.g., recursively and iteratively, should produce the same call to the Divide and Conquer skeleton. Again, this is ongoing and future work.

Chapter 7

Caveats

There are a number of problems which we believe are due to bugs in CDT and/or GrPPI.

7.1 General

- GrPPI requires g++ >= 6.2.
- GrPPI does not currently work with Clang. It appears that some of the template code used in GrPPI does not adhere strictly to the C++ specification, and Clang rejects it because of this.
- We experienced some difficulty involving relocation errors at runtime with both g++ 6.2 and g++ 7.1. This appears to be due to incompatibility with our system version of glibc. We were able to work around this by adding the compilation flag `-D_GLIBCXX_USE_CXX11_ABI=0` as suggested in https://stackoverflow.com/questions/36816570/, although it is not clear to us whether this is entirely safe.

7.2 Spurious template errors in CDT

The CDT interface sometimes reports problems in the refactored code, especially when farm stages are involved (see the figure below).

Despite the purported errors, the refactored code can still be built and run successfully, even within Eclipse.

We have been unable to work around this, and we believe that it is due either to the GrPPI bug mentioned above or to limitations of the CDT parser when dealing with complex template code.

Figure 7.1: Spurious template errors

7.3 Incorrectly rendered types

The plugin occasionally has difficulty producing the expected representation of types in refactored code: for example, we sometimes produce types like `std::vector<int, allocator<int>>::iterator` where `vector<int>::iterator` would suffice (and note also that `allocator` should be `std::allocator`). This is because we are using internal functions from the Eclipse/CDT infrastructure which sometimes appears to have difficulty rendering types properly, possibly due to bugs in CDT (see https://bugs.eclipse.org/bugs/show_bug.cgi?id=513063 for something similar).

It is possible (although we have not seen this in practice so far) that particularly complex types may be rendered in refactored code in a way which is unacceptable to the C++ parser. In such cases some manual rewriting of types may be necessary; adding `using namespace std;` may also help in some cases.

Chapter 8

Conclusion

In this deliverable we described our final refactorings that target the GrPPI framework, including refactoring to introduce a pipeline, map reduce, divde and conquer and stencil patterns in GrPPI. The refactorings are implemented in the Eclipse refactoring tool developed by USTAN, in the CDT framework and support C++14 features.

The Introduce Pipeline refactoring described in Chapter 2 extends the refactoring first introduced in D2.1. The extensions described here facilitate the application of the refactoring to more general code, including experimental support for `while` loops. The Introduce MapReduce and Introduce Stencil refactorings, described in Chapters 4 and 5, transform pragma-annotated `for` loops that iterate over vectors into calls to map reduce and stencil skeletons. The Introduce Divide and Conquer skeleton, described in Chapter 6, transforms a pragma-annotated recursive function declaration into a non-recursive function that calls the GrPPI divide and conquer skeleton. The refactorings to introduce map reduce, stencil, and divide and conquer patterns illustrate how these patterns can be introduced (semi-)automatically. Future work will facilitate the refactoring of more general code, e.g. iterative approaches to divide and conquer, in addition to greater automation, e.g. liveness analysis to automatically detect variables that need to be returned by individual skeleton stages. A list of caveats to the refactoring implementations was given in Chapter 7.

Appendix A

Pipeline example

We give a simple (and unrealistic) example of a loop equipped with pragmas, and show how the pipeline refactoring transforms it. Some explanatory comments have been added to the refactored code.

Original program

```
#include <iostream>
#include <cmath>

using namespace std;

int main(int argc, char **argv) {
   constexpr int N = 100000;

   double A[N], B[N];
   for (int i = 0; i < N; ++i) {
      A[i] = 2*i+1;
   }

   for (int j = 0; j < N; ++j) {
      double x = sqrt(A[j]);
      x+=1;
      B[j] = x;
      cout << x << endl;
   }
}
```

Program after annotation

```
#include <iostream>
#include <cmath>

using namespace std;

int main(int argc, char **argv) {
   constexpr int N = 100000;

   double A[N], B[N];
   for (int i = 0; i < N; ++i) {
      A[i] = 2*i+1;
   }
```

```
   for (int j = 0; j < N; ++j) {
#pragma grppi seq stage
      double x = sqrt(A[j]);
#pragma grppi farm stage 16
// These instructions will be executed in parallel on multiple data items
      x+=1;
      B[j] = x;
#pragma grppi seq stage
      cout << x << endl;
   }
}
```

Program after refactoring

```
#include <iostream>
#include <cmath>

// Following headers inserted by tool
#include <vector>

#include "grppi.h"
#include "seq/sequential_execution.h"
#include "native/parallel_execution_native.h"
#include "omp.h"
#include "omp/parallel_execution_omp.h"
#include "tbb/parallel_execution_tbb.h"

using namespace grppi;
// Inserted by tool: preference allows use of qualified names

using namespace std;

int main(int argc, char **argv) {
   constexpr int N = 100000;

   double A[N], B[N];
   for (int i = 0; i < N; ++i) {
      A[i] = 2*i+1;
   }

   int j = 0; // Variable declaration moved outside loop
   parallel_execution_native pipe { };
   // Change this to change the parallelism model

   pipeline(pipe,
   // -------- source ----------
         [&]() {  // Source lambda to return sequence of values of j
            if (j < N) {
               auto _r = j;
               ++j;
               return std::experimental::optional<int>(_r);
            } else return std::experimental::optional<int>();
         },
         // -------- stage 1 ----------
         [&](int j) {  // Value of j forwarded from source
            double x = sqrt(A[j]);  // External variable A captured by reference
            return std::make_tuple(x, j);  // Return j and x for next stage
         },
         // -------- stage 2 ----------
         farm(16, [&](std::tuple<double, int> _args) {
            double x = std::get<0>(_args); // Unpack values from tuple
            int j = std::get<1>(_args);
            x+=1;
            B[j] = x;
            return x;  // j no longer needed
         }),
         // -------- stage 3 ----------
         [&](double x) {
            cout << x << endl;
            return;
         });
}
```