Project no. 288570

PARAPHRASE

Strategic Research Partnership (STREP)

PARALLEL PATTERNS FOR ADAPTIVE HETEROGENEOUS MULTICORE SYSTEMS

Final Pattern Transformation System

D4.4

Due date of deliverable: 31 March 2015

Start date of project: October 1st, 2011

Type: Deliverable
WP number: WP4
Task number: T4.1

Responsible institution: University of St Andrews

Editor and editor’s address: Christopher Brown, University of St Andrews

Project co-funded by the European Commission within the Seventh Framework Programme

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>Public</td>
</tr>
<tr>
<td>PP</td>
<td>Restricted to other programme participants (including the Commission Services)</td>
</tr>
<tr>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including the Commission Services)</td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including the Commission Services)</td>
</tr>
</tbody>
</table>

Version 0.1 ()
Executive Summary

This deliverable describes the final pattern transformation system, where we describe a number of new advancements, including:

- the LAPEDO framework, a new refactoring framework for introducing heterogeneous (i.e., a mixture of CPU and GPU components) skeletons into C++ (using Fastflow) and Erlang (using Skel) programs; this combines hybrid skeletons for C++/Erlang, program shaping refactorings, automatic generation of openCL offloading code, a static division of work between CPU and GPU devices, and new refactorings to introduce hybrid skeletons into C++/Erlang programs;

- PaRTE (Parallel Refactoring Tool for Erlang), incorporating the Erlang refactoring implementation from D4.1 and the pattern discovery from D2.11, D2.12 and D.13 into an integrated tool set for parallelising Erlang programs;

- Formalising Refactorings, using an operational and functional semantics to show soundness;

- Refactoring use-cases for heterogeneous examples in Erlang and also demonstrating Program Shaping on the AGH use-case.

Positioning of Deliverable 4.4

The positioning of this deliverable (D4.4) with respect to other deliverables is shown in Figure 1. In particular, the work presented in D4.4 is based upon the work presented in:

- D2.9 (Final Heterogeneous Implementation: Software Implementation of Final Patterns for Heterogeneous Multicore Architectures);

- D3.2 (Static Mapping Implementation: Implementation of adaptive mapping of components to hardware resources);

- D4.5 (Technical report on pattern candidate refactoring rules); and,

- D6.9 (Report on MAS Experimental Evaluation).
Figure 1: The Positioning of D4.4
Contents

Executive Summary ........................................................................................................ 1
Positioning of Deliverable 4.4 .................................................................................... 1

1 Introduction ................................................................................................................ 5
    1.1 What is new in this deliverable? ........................................................................... 5

2 Formalisation of Refactorings .................................................................................... 7
    2.1 Formalising Rewritings ....................................................................................... 7
        2.1.1 Denotational Semantics of Skeletons ....................................................... 7
        2.1.2 Functional Correctness of the Denotational Semantics. ....................... 10
        2.1.3 Rewriting Skeletal Expressions ............................................................. 11
    2.2 Soundness of the Refactorings ........................................................................... 12
        2.2.1 Functional Semantics of Erlang Skeletons ............................................. 12
        2.2.2 Proving Correctness of Erlang Skeletons .............................................. 15

3 The Lapedo Framework for C++ and Erlang ............................................................. 17
    3.1 Lapedo Example in C++ .................................................................................... 19
    3.2 Lapedo Example in Erlang ................................................................................. 27
    3.3 GPU Programming ............................................................................................ 33
        3.3.1 GPU Programming in Erlang ................................................................. 33
    3.4 Hybrid Skeletons in Lapedo .............................................................................. 33
        3.4.1 Hybrid Skeletons in C++/Fastflow ......................................................... 33
        3.4.2 Hybrid Skeletons in Erlang ................................................................. 33
        3.4.3 Division of Work Between CPUs and GPUs. ....................................... 35
    3.5 Lapedo Refactorings for Hybrid Skeletons and Program Shaping ..................... 37
        3.5.1 Refactorings to Introduce Hybrid Skeletons ........................................ 37
        3.5.2 Refactorings for Program Shaping ...................................................... 38
        3.5.3 Generating GPU Worker Code (Offloading) ...................................... 39
    3.6 Applying Lapedo to C++ ................................................................................... 41
    3.7 Heterogeneous Refactoring Examples ............................................................. 42
        3.7.1 Examples in Fastflow/C++ ..................................................................... 42
        3.7.2 Examples in Erlang ............................................................................ 45
# ParaPhrase Refactoring Tool for Erlang

4.1 The PaRTE framework ........................................ 55
4.2 Ranking with cost analysis ................................... 56
   4.2.1 Compound pattern candidates .......................... 56
   4.2.2 Measuring and calculating sequential execution time .......................... 57
   4.2.3 Calculating parallel execution time ......................... 61
4.3 DSL for refactorings and rule interpretation .................... 61
   4.3.1 The Refactoring Language ............................... 62
   4.3.2 Examples .................................................. 64
   4.3.3 Rule Interpreter .......................................... 65

# A Refactoring Example: Agent Systems

5.1 New Shaping: Extract Comprehension Function .................. 68
   5.1.1 Motivation ................................................. 68
   5.1.2 Description ............................................... 68
   5.1.3 Conditions ................................................ 68
5.2 New Shaping: Compose Maps ................................... 69
   5.2.1 Motivation ................................................. 69
   5.2.2 Description ............................................... 69
   5.2.3 Conditions ................................................ 69
5.3 Refactoring the Agent System .................................. 69
   5.3.1 Stage 1 .................................................... 69
5.4 Stage 2 .......................................................... 70
5.5 Stage 3 .......................................................... 71
5.6 Stage 4 .......................................................... 72
5.7 Stage 5 .......................................................... 74
5.8 Stage 6 .......................................................... 74
5.9 Stage 7 .......................................................... 75
5.10 Using Binaries .................................................. 76
5.11 Results .......................................................... 77

# Conclusion

6 ................................................................. 80

# A Full list of Refactoring Rules

A.1 List to Binary ............................................... 83
A.2 List or Binary to ETS Table ................................... 86
A.3 Introduce Hybrid Map Rules ................................... 87
A.4 Parallel Map Introduction ..................................... 87
A.5 Introduce Task Farm Refactoring ............................... 88
Chapter 1

Introduction

This deliverable describes the final phase of the Refactoring Tools implemented for the Pattern Transformation System, as described in T4.1. The work here focuses on providing heterogeneous refactoring support for both C++ and Erlang. The work here builds heavily on the heterogeneous skeleton work as proposed in D2.4 and D2.9 for C++ and Erlang and also the MAS Use-Cases as reported in D6.9.

D4.1 reported an initial transformation system, targeted at Erlang demonstrating a number of new refactorings that introduce and tune parallelism, such as Introduce Pipeline, Introduce Farm, Introduce Map and Introduce Chunking. D4.2 extended this initial set with new refactoring implementations for C++ built into Eclipse. In this deliverable we extend this refactoring set with new hybrid refactorings that introduce heterogeneous skeletons in C++ and Erlang programs, via our LAPEDO framework. We demonstrate LAPEDO on a number of use-cases in C++ and Erlang, including Image Convolution, Molecular Dynamics (provided by HLRS), and Football Simulation (provided by Erlang Solutions). We have extended our formal work with operational and functional semantics for skeletons, and shown how to prove soundness of our refactorings w.r.t. to these semantics. This is important to guarantee that our refactorings are safe and valid, and won’t introduce any undesired behaviour into the application. We show our soundness on Erlang because Erlang is simpler than C++ to describe in terms of its functionality. However, our soundness principle can be easily extended to C++ in the future. We introduce PaRTE, our Erlang refactoring framework that also includes pattern discovery and a refactoring DSL for describing simple, high-level refactorings for Erlang. Finally, we show how the Agent Systems use-case from AGH can be fully refactored using our system.

1.1 What is new in this deliverable?

- We give an operational and denotational semantics for some simple skeletons and show, via these semantics a soundness proof for an example refactoring;
- We the introduce LAPEDO framework, allowing programmers to introduce...
heterogeneous skeletons into their C++/Erlang applications using a combination of refactoring, hybrid skeletons, automatic code-offloading generation and static work allocation;

- We introduce hybrid skeletons, that allow a mixture of CPU and GPU components to be integrated into a skeletal system. The Hybrid skeletons also have an automatic static work allocator that allocates an optimal ratio of tasks to each CPU and GPU component;

- We introduce hybrid refactorings that introduce hybrid skeletons into Erlang programs;

- We introduce program shaping, a new concept of re-structuring a sequential application into the appropriate shape so that it is possible to introduce skeletons via refactoring;

- We introduce the PaRTE tool, a combination of our Erlang refactorings, pattern discovery, refactoring rules and refactoring DSL;

- We present a series of heterogeneous use cases in C++ and Erlang, demonstrating the effectiveness of our LAPEDO framework.

- We demonstrate our refactoring tool used by our industrial partner, AGH, on a use case from their agent-based systems.
Chapter 2

Formalisation of Refactorings

In this chapter we derive a formal approach to refactoring. We start by first introducing a denotational semantics for some simple skeletons, and then we show functional correctness of a simple rewriting system based on our refactorings in D4.1. The main idea is that any valid rewriting applied to a parallel program should preserve the denotational semantics, i.e. it must not change its functional behaviour. This is an important property, since it allows us to ensure that whenever we transform a program according to these rules, we will not introduce any new bugs in it. We finish this chapter by showing the soundess of an Erlang refactoring (Introducing Farm) in a similar way, by first describing our Erlang skeletons in terms of their functional semantics, and then using this semantics to show that introducing a Farm skeleton into an Erlang program preserves its functional semantics.

2.1 Formalising Rewritings

We show a formalisation of rewritings that change the operational (parallel) behaviour of a structured parallel program described in terms of algorithmic skeletons. The proofs of the main lemmas and theorems shown here were done in the IDRIS dependently typed functional language, which can be accessed online in https://bitbucket.org/david_castro/idris_skel. We also provide in this repository a draft paper, which is currently under submission for the Journal of Functional Programming, describing in more detail the proofs and technique (https://bitbucket.org/david_castro/idris_skel/downloads/paper.pdf).

2.1.1 Denotational Semantics of Skeletons

Our denotational semantics operates on values in the domain of skeletal expressions, $S$. defined below. $V$ is the domain of values (which we will leave partly unspecified), $V \rightarrow V$ represents functions over those values, and $V^*$ represents sequences of values (including lists and streams). $\Sigma$ is the domain of skeleton
structures, corresponding to $S$.

$$n \in \mathbb{N} = \{0, 1, \ldots\} \quad \text{natural numbers}$$

$$v, x \in \mathbb{V} = \ldots | \mathbb{F} | \mathbb{L} \quad \text{values}$$

$$f \in \mathbb{F} = \mathbb{V} \to \mathbb{V} \quad \text{functions}$$

$$fs \in \mathbb{L} = \mathbb{V}^* \quad \text{sequences}$$

$$s \in S ::= \text{func} f | \text{farm} n s | \text{pipe} s s | \text{seq} s s \quad \text{skeletal expressions}$$

$$\sigma \in \Sigma ::= \text{Func} | \text{Farm} n \sigma | \text{Pipe} \sigma \sigma | \text{Seq} \sigma \sigma \quad \text{skeleton structure}$$

As an example, consider the following Fastflow example from the Fastflow online tutorial. It consists of a 3-stage pipeline with a farm in the last stage. The workers are, respectively firstStage, secondStage and thirdStage:

```cpp
#include <ff/farm.hpp>
#include <ff/pipeline.hpp>
using namespace ff;
typedef long fftask_t;
struct firstStage: ff_node_t<task_t> {
    fftask_t *svc(fftask_t *t) {
        ...
    }
};
fftask_t* secondStage(fftask_t *t, ff_node* const node) {
    ...
}
struct thirdStage: ff_node_t<task_t> {
    fftask_t *svc(fftask_t *t) {
        ...
    }
};
int main() {
    std::vector<ff_node*> W = { new thirdStage , new thirdStage };
    ff_pipe<fftask_t> pipe( new firstStage , secondStage , new ff_farm<>(W));
    pipe.cleanup_nodes();
    if (pipe.run_and_wait_end()<0)
        error("running pipe");
    return 0;
}
```
The same example can be represented in our formalism as shown:

\[
\text{prog} = \text{pipe (func fistStage) (pipe (func secondStage) (farm 2 (func thirdStage)))}
\]

\[
\text{prog}_\sigma = \text{Pipe Func (Pipe Func (Farm 2 Func))}
\]

\[
\text{prog}_{fs} = \langle \text{firstStage, secondStage, thirdStage} \rangle
\]

The definition of \text{prog} is a 3-stage pipeline equivalent to the Fastflow example, built as the composition of two 2-stage pipelines, where the last stage is a farm with two workers. We provide also the corresponding structure, \text{prog}_\sigma, and the functionality, \text{prog}_{fs}.

The denotational semantics of a skeletal expression, \( S \), is defined by \( J_S K \). This yields a value of type \( V^* \rightarrow V^* \), that maps a stream of input values to a stream of results:

\[
[J_S] : V^* \rightarrow V^*
\]

\[
[J_{\text{func } f}] = \text{map } f
\]

\[
[J_{\text{pipe } s_1 s_2}] = [s_2] \circ [s_1]
\]

\[
[J_{\text{seq } s_1 s_2}] = [s_2] \circ [s_1]
\]

\[
[J_{\text{farm } n s}] = \text{perm} \circ [s]
\]

Here, \( \circ : (V \rightarrow V) \rightarrow (V \rightarrow V) \rightarrow V \rightarrow V \) denotes the pairwise composition of two functions, \( \text{map} : (V \rightarrow V) \rightarrow V^* \rightarrow V^* \) applies a function to all the elements in a list, and \( \text{perm} : V^* \rightarrow V^* \) arbitrarily permutes a list of elements, so allowing farm results to be returned in an arbitrary order, to allow the option of a more efficient implementation, as described above. We can now define some auxiliary relations that will allow us to state strong properties about \([S]\).

**Definition 2.1.1 (Structural Equivalence Relation)** The \( \sim_{fs} \) relation relates values in the domains of \( S \) and \( \Sigma \) via the list of functions \( fs \in (V \rightarrow V)^* \). \( s \sim_{fs} \sigma \) indicates that \( s \) uses the functions \( fs \) in order in the structure \( \sigma \).

\[
\begin{align*}
\text{func } f & \sim_{(f)} \text{ Func} \\
\text{pipe } s_1 s_2 & \sim_{fs_1 \oplus fs_2} \text{ Pipe } \sigma_1 \sigma_2 \\
\text{seq } s_1 s_2 & \sim_{fs_1 \oplus fs_2} \text{ Seq } \sigma_1 \sigma_2
\end{align*}
\]

Here, \( \oplus \) denotes the concatenation of two lists. Given \( s \in S \), \( \sigma \in \Sigma \) and \( fs \in (V \rightarrow V)^* \), such that \( s \sim_{fs} \sigma \), \( fs \) directly determines the denotational semantics of \( s, [s] \), as shown by Theorem 2.1.1.

**Definition 2.1.2 (Reverse Composition)** The reverse composition of a list of functions, \( \circ : (V \rightarrow V)^* \rightarrow V \rightarrow V \) is defined as: \( \circ \langle f_1, \ldots, f_n \rangle = f_n \circ \ldots \circ f_1 \)
2.1.2 Functional Correctness of the Denotational Semantics.

We are now able to state a key soundness result for the denotational semantics. Theorem 2.1.1 states that the denotational semantics of any \( s \in S \) is some valid permutation of the result of applying the functions that are contained within \( s \). That is, the functionality of \( s \) fully determines its results, but in some arbitrary order. This allows the possibility of using one or more farms. Given \( s \in S \), we define the functions of \( s \), \( \text{funcs}(s) = f_s \), such that \( \exists \sigma \in \Sigma, s \leadsto f_s \sigma \).

**Theorem 2.1.1** For all \( s \in S \), \( \llbracket s \rrbracket = \text{perm} \circ \text{map}(\bigcirc(\text{funcs}(s))) \).

In the Fastflow example the following holds: \( \text{prog} \leadsto \text{prog}_{\text{prog}_{\sigma_{\text{prog}}}} \). That is, \( \text{prog} \) has structure \( \text{prog}_{\sigma} \) with functional behaviour determined by the functions \( \text{prog}_{f_s} \). This implies, as expected, that the functional behaviour of the skeleton is \( \text{perm} \circ \text{map}(\text{thirdStage} \circ \text{secondStage} \circ \text{firstStage}) \).

**Corollary 2.1.1** For all \( f_s \in (V \rightarrow V)^* \), \( s \in S \) and \( \sigma \in \Sigma \), such that \( s \leadsto f_s \sigma \), \( \llbracket s \rrbracket = \text{map}(\bigcirc f_s) \) iff \( s \) contains no \( \sigma \text{-farm} \) subexpressions (transitively).

**Definition 2.1.3 (Functional Equivalence)** We say that any two skeletal expressions \( s_1, s_2 \in S \) are functionally equivalent iff \( \llbracket s_1 \rrbracket = \llbracket s_2 \rrbracket \).

Using Theorem 2.1.1, if \( \sigma_1, \sigma_2 \in \Sigma \) and \( f_{s_1}, f_{s_2} \in (V \rightarrow V)^* \) such that \( s_1 \leadsto f_{s_1} \sigma_1 \) and \( s_2 \leadsto f_{s_2} \sigma_2 \), \( \llbracket s_1 \rrbracket = \llbracket s_2 \rrbracket \) is equivalent to \( \text{perm} \circ \text{map}(\bigcirc f_{s_1}) = \text{perm} \circ \text{map}(\bigcirc f_{s_2}) \). Since \( \text{perm} \) denotes any arbitrary permutation, we cannot guarantee that both sides of the equality return the same output in the same order unless we use Corollary 2.1.1 and avoid using task farms. We can, however, weaken this condition by defining equality modulo permutations.

**Definition 2.1.4 (Equality Modulo Permutations)** For all \( xs, ys \in V^* \) such that \( xs = \text{perm}(ys) \), we say that \( xs =_{P} ys \).

It is easy to show that \( =_{P} \) is reflexive, symmetric and transitive. We will abuse this notation slightly and also write \( f =_{P} g \) instead of \( \forall xs \in V^*, f(xs) =_{P} g(xs) \). This allows us to define a weaker notion of functional equivalence.

10
Definition 2.1.5 (Weak Functional Equivalence) Any two skeletal expressions \( s_1, s_2 \in S \) are weakly functional equivalent iff \( \llbracket s_1 \rrbracket =_p \llbracket s_2 \rrbracket \).

This definition allows us to determine the (weak) equivalence of two skeletons.

Theorem 2.1.2 (Weak Functional Equivalence of Skeletons) For all \( s_1, s_2 \in S \), if \( \text{funcs}(s_1) = \text{funcs}(s_2) \), then \( \llbracket s_1 \rrbracket =_p \llbracket s_2 \rrbracket \).

An important remark about Theorem 2.1.2 is that if \( \text{funcs}(s_1) \neq \text{funcs}(s_2) \), we cannot claim anything about the equivalence of \( s_1 \) and \( s_2 \), since \( \bigcirc f_{s_1} \) might be equal to \( \bigcirc f_{s_2} \) even if both lists of functions are not (syntactically) equal. It is, however, reasonable to keep the condition \( f_{s_1} = f_{s_2} \) at a syntactic level, since determining the equivalence of two arbitrary functions is an undecidable problem.

2.1.3 Rewriting Skeletal Expressions

Any tree of algorithmic skeletons can be rewritten into an alternative parallel structure, provided that it preserves its denotational semantics, i.e. it is possible to use any two \( s_1, s_2 \in S \) interchangeably, provided that \( \llbracket s_1 \rrbracket = \llbracket s_2 \rrbracket \). Theorem 2.1.2 guarantees that whenever the \( f_s \) are the same, the functional behaviour will be the same (modulo permutations). However, it would be more useful to define an equivalence relation, \( \equiv \in \Sigma \times \Sigma \), that defines families of (weakly) functionally equivalent skeletal expressions. The convertibility relation captures some of the refactorings that from deliverable D4.1 and D4.2 (farm and pipeline introduction/elimination), and properties of function composition (associativity).

Definition 2.1.6 (Convertibility Relation)

\[
\begin{align*}
\text{REFL} & : \sigma \equiv_{Cnv} \sigma \\
\text{SYM} & : \sigma_1 \equiv_{Cnv} \sigma_2 \Rightarrow \sigma_2 \equiv_{Cnv} \sigma_1 \\
\text{TRANS} & : \sigma_1 \equiv_{Cnv} \sigma_2 \quad \sigma_2 \equiv_{Cnv} \sigma_3 \\
\sigma_1 & \equiv_{Cnv} \sigma_3 \\
\text{SEQASSOC} & : \text{Seq}(\text{Seq} \sigma_1 \sigma_2) \equiv_{Cnv} \text{Seq} \sigma_1 (\text{Seq} \sigma_2 \sigma_3) \\
\text{PIPEINTRO} & : \text{Seq} \sigma_1 \sigma_2 \equiv_{Cnv} \text{Pipe} \sigma_1 \sigma_2 \\
\text{RSEQ} & : \text{Seq} \sigma_1 \sigma_2 \equiv_{Cnv} \text{Seq} \sigma_3 \sigma_4 \\
\text{FARMINTRO} & : n_1 \in \mathbb{N} \\
\sigma & \equiv_{Cnv} \text{Farm} n_1 \sigma
\end{align*}
\]

\(^1\)This is derived from a set of rules that are well-known in the parallel programming literature as “structural equivalences” \([1, 2, 8]\). We prefer the term “convertibility” here, since the relation determines whether we can convert a skeletal expression that has one structure into a functionally equivalent one that has a different structure.
We define a normalisation function \( \text{norm} : \Sigma \rightarrow \Sigma \) such that for all \( \sigma \in \Sigma \), \( \text{norm}(\sigma) \) contains no parallel structure and all the occurrences of \( \text{Seq} \) are right-associative. This function can be defined as the composition of two other functions \( \text{norm} = \text{flatten} \circ \text{rmpar} \), where:

- \( \text{rmpar} : \Sigma \rightarrow \Sigma \)
  - \( \text{rmpar}(\text{Func} \ \sigma_1 \ \sigma_2) = \text{Func} \)
  - \( \text{rmpar}(\text{Pipe} \ \sigma_1 \ \sigma_2) = \text{Seq}(\text{rmpar}(\sigma_1)) \ (\text{rmpar}(\sigma_2)) \)
  - \( \text{rmpar}(\text{Farm} \ n \ \sigma) = \text{rmpar}(\sigma_1) \)

- \( \text{flatten} : \Sigma \rightarrow \Sigma \)
  - \( \text{flatten}(\text{Seq}(\text{Seq} \ \sigma_1 \ \sigma_2) \ \sigma_3) = \text{flatten}(\text{Seq} \ \sigma_1 \ (\text{Seq} \ \sigma_2 \ \sigma_3)) \)
  - \( \text{flatten}(\text{Seq} \ \sigma_1 \ \sigma_2) = \text{Seq} \ \sigma_1 \ (\text{flatten}(\sigma_2)) \)
  - \( \text{flatten}(\sigma_1) = \sigma_1 \)

**Lemma 2.1.1** For all \( \sigma \in \Sigma \), \( \sigma \equiv_{\text{Cnv}} \text{norm}(\sigma) \).

**Lemma 2.1.2** For all \( \sigma_1, \sigma_2 \in \Sigma \), \( \sigma_1 \equiv_{\text{Cnv}} \sigma_2 \iff \text{norm}(\sigma_1) = \text{norm}(\sigma_2) \).

**Theorem 2.1.3** For all \( s_1 \in S \) and \( \sigma_1, \sigma_2 \in \Sigma \) such that \( s_1 \sim_{f} \sigma_1, \sigma_1 \equiv_{\text{Cnv}} \sigma_2 \iff \exists s_2 \in S \) such that \( s_2 \sim_{f} \sigma_2 \).

### 2.2 Soundness of the Refactorings

Showing a soundness of a refactoring means to prove that the refactoring does what it is supposed to do. In this section we show how it is possible to show the soundness of some of the refactorings described in previous deliverables (D4.1 and D4.2). In particular, in this section we show a soundness example for Erlang. As Erlang is a functional programming language it is easier to show soundness as it is easy to specify the Erlang skeletons in terms of their functional semantics. However, the techniques shown here can easily be transferred to C++. The functional semantics of the skeletons is described in terms of sequential Erlang functions that describe what the different skeletons compute (rather than how), and the correctness of the refactorings is proven by showing the correspondence of the functional semantics of equivalent parallel structures. This functional semantics is actually the specification of any valid skeleton library implementation, i.e. the semantics of any real implementation of skeletons must be sound with respect to the functional semantics described here.

#### 2.2.1 Functional Semantics of Erlang Skeletons

We define the following functional semantics for our skeleton implementation. We note that whilst our semantics and implementation use queues and lists, these are
functionally equivalent to partial application of curried functions. In each of the below definitions we therefore treat all queues as individual elements from those queues.

\[
\text{spawn } s \ q_{in} = \\
\lambda x. (s \ x)(q_{in}).
\]

Spawn takes as parameter a skeleton configuration, applying each element from the input queue to that configuration as if by partial evaluation.

\[
\text{send } x \ q_{in} = \\
(q_{in} \ x).
\]

Send applies a given input to a queue, thereby enqueueing it.

\[
\text{receive } q_{out} = \\
q_{out}.
\]

Receive returns elements from a queue, thereby dequeueing them.

\[
\text{skeldo } s \ xs = \\
q_{in} = \text{spawn } s, \\
\text{receive } (\text{skelem } q_{in} \ xs).
\]
Representative of the top-level Skel function, \texttt{skeldo} takes a skeleton configuration and a list of inputs. It sets up the configuration, and sends each element in the input list through the configuration, eventually returning the result of the configuration when applied to each element in the list.

\begin{verbatim}
skeldo qin xs =
    map qin xs.
\end{verbatim}

The emitter function used to enqueue each element of the input list, allowing those elements to pass through the skeleton configuration given to \texttt{skeldo}.

\begin{verbatim}
func f qin =
    \ x. (f x) (qin).
\end{verbatim}

Representative of the \texttt{func} skeleton in Skel; applies input to a programmer-provided function.

\begin{verbatim}
farm s n qin =
    receive (farmem (replicate (spawn s) n) (send qin))
\end{verbatim}

Representative of the \texttt{farm} skeleton in Skel; generates a given number of workers which apply a given skeleton configuration to each input.

\begin{verbatim}
farmem (w_1, \ldots, w_n) qin =
    let (qin_1, \ldots, qin_n) = split (replicate n []) qin in:
        ((send qin_1 w_1), \ldots, (send qin_n w_n))
\end{verbatim}

The emitter function used to enqueue inputs to a finite number of workers.

\begin{verbatim}
split ws xs =
    foreach x in xs:
        ws[(pos x) mod n] = ws[(pos x) mod n] ++ x
\end{verbatim}

A function assigning inputs to workers.
2.2.2 Proving Correctness of Erlang Skeletons

By using equational reasoning we show the functional equivalence of the sequential Erlang map operation applying \( f \) to inputs \( xs \), and the invocation of Skel with a skeleton configuration of a single \texttt{func} applying \( f \) to inputs, and input of \( xs \).

\[
\text{skeldo \ (func \ f \) \ xs}.
\]

\[
q_{in} = \text{spawn \ (func \ f)},
\]

\[
\text{receive \ (skelem \ q_{in} \ xs)}
\]

\[
q_{in} = \text{spawn \ (} \backslash x.\ (f\ x)\text{)},
\]

\[
\text{receive \ (skelem \ q_{in} \ xs)}
\]

\[
q_{in} = \backslash y.\ (\backslash x.\ (f\ x)\ (y)),
\]

\[
\text{receive \ (skelem \ (} \backslash y.\ (\backslash x.\ (f\ x)\ (y))\text{) \ xs)}.
\]

\[
\text{receive \ (map \ (} \backslash y.\ (\backslash x.\ (f\ x)\ (y))\text{) \ xs)}.
\]

Which, by \( \eta \) reduction:

\[
\text{receive \ (map \ (} \backslash x.\ (f\ x)\text{) \ xs)}.
\]

\[
\text{receive \ (map \ f \ xs)}.
\]

\[
\text{map \ f \ xs}.
\]

We next show the equivalence of Skel with a skeleton configuration of a farm of \( n \) workers, with nested configuration of a \texttt{func} applying \( f \) to inputs, taking \( xs \) as input with a Skel invocation with skeleton configuration of a \texttt{func} applying \( f \) to inputs, taking \( xs \) as input; thereby showing by transitivity its equivalence to a map operation applying \( f \) to \( xs \).

\[
\text{skeldo \ (farm \ n \ (func \ f)) \ xs}.
\]

\[
\text{receive \ (skelem \ (spawn \ (farm \ n \ (func \ f))) \ xs)}
\]

\[
\text{skelem \ (spawn \ (farm \ n \ (func \ f))) \ xs}
\]

\[
\text{map \ (spawn \ (farm \ n \ (func \ f))) \ xs}
\]

15
\begin{align*}
\text{map } (\lambda x. (\text{farm } n \ (\text{func } f) \ x)) \ xs \\
\text{map } (\lambda x. (\text{farmem} \ (\text{replicate} \ (\text{func } f) \ n) \ (\text{send } x))) \ xs
\end{align*}

Assuming \( f \) pure, and as \text{replicate} \ (\text{func } f) \ n \ produces \ a \ list \ of \ n \ identical \ workers, \ and \ where \ \text{farmem} \ ws \ xs \ cycles \ through \ each \ of \ the \ workers \ when \ applying \ from \ xs, \ \text{we \ can \ assume \ the \ operation} \\
\lambda y. \ \text{farmem} \ (\text{replicate} \ (\text{func } f) \ n) \ (\text{send } y) \ \equiv \lambda y. (\text{send } y \ (\text{func } f))

\begin{align*}
\text{map } (\lambda x. (\text{send } x \ (\text{func } f))) \ xs \\
\text{map } (\lambda x. (\lambda y. (\text{func } f) \ y)) \ xs \\
\text{map } (\lambda x. (\lambda y. (\lambda z. f \ z) \ y)) \ xs
\end{align*}

Which, by \( \eta \) reduction:

\begin{align*}
\text{map } (\lambda y. (\lambda z. (f z)(y))) \ xs \\
\text{map } (\lambda z. (f z)) \ xs \\
\text{map } f \ xs
\end{align*}
Chapter 3

The Lapedo Framework for C++ and Erlang

In this chapter we introduce LAPEDO, a framework for programming heterogeneous multi-core systems in C++/Erlang, using refactoring and skeletons. LAPEDO aims to guide the programmer through the process of converting a sequential Erlang program into an equivalent parallel version that can be executed efficiently on a heterogeneous multicore system. While LAPEDO primarily targets CPU+GPU combinations, it can also, in principle, be used for other kinds of heterogeneous system that support the OpenCL model (e.g. ones with FPGAs and/or Xeon Phis as well). LAPEDO consists of: i) The C++ Paraphrase refactoring tool (described in D4.2) and the Erlang Paraphrase refactoring tool (PaRTE as described in Chapter 4 and Deliverable D4.1), enhanced with refactorings for parallelism introduction and program shaping (Section 3.5); ii) the FastFlow (D2.2) and the Skel library (D2.2), enhanced with hybrid skeletons that have both CPU and GPU components (Section 3.4); and iii) a GPU Code Generation tool that generates the control logic for the GPU computations (Section 3.5.3). Figure 3.1 provides an overview of the process that is involved in transforming an existing C++/Erlang application into a parallel equivalent using LAPEDO. The programmer starts with a sequential CPU application and an OpenCL kernel that provides a GPU implementation for suitable operations in the application. The programmer then performs the following steps (which can be repeated if necessary):

1. Introducing CPU-only skeletons. The programmer first uses the ParaPhrase C++ Eclipse Refactoring tool (from D4.2) or PaRTE (described in Chapter 4) to parallelise code for CPUs only, using the appropriate skeletons from FastFlow (in C++) or Skel (in Erlang) and the parallelism-introduction refactorings that introduce CPU skeletons, described in D4.1 (for Erlang) and D4.2 (for C++). This step may involve some program shaping transformations on the sequential code, to prepare the code for the introduction of parallelism (some of these refactorings are specific to Erlang due to the fact, in Erlang, GPU devices must receive data in Erlang’s binary format). This gives us a
base parallelisation, on which we can use further program-shaping refactorings and the refactorings that introduce hybrid skeletons.

2. **Program shaping.** In the next step, the programmer applies any necessary program shaping transformations. For example, as described in Section 3.3.1, the basic Erlang OpenCL bindings only support the transfer of binary data to the GPU, so any offloaded lists must be converted into binaries, together with the associated operations. Additionally, we may want to improve the CPU-only parallel version of the code, by using more appropriate data structures, such as ETS tables. A C++ example of a program shaping refactoring, may include refactoring the arguments of a function into a C++ Structure. These, and other program shaping transformations, are described in more detail in Section 3.5.2. *This step is semi-automatic,* requiring the programmer to select the regions of the code that need to be transformed, and to choose the appropriate transformations;

3. **GPU code generation.** In the next step, the programmer invokes the GPU Code Generator to generate C++/Erlang code for the GPU components of the future hybrid application. This generates code to link with and execute GPU kernels (e.g. to manage data transfers to/from the GPU and to schedule the GPU computations), based on programmer-supplied information about
the kernels. The GPU Code Generator is described in more detail in Section 3.5.3. This step is semi-automatic, requiring the programmer to supply information about the GPU kernels;

4. Introduction of hybrid patterns. Finally, the programmer uses the C++ refactoring tool/ParTE again to refactor the shaped CPU code in order to introduce hybrid skeletons. This results in a hybrid application that uses the GPU components that have been generated in the previous step. Hybrid skeletons are described in more detail in Section 3.4, and the refactorings that introduce them are described in Section 3.5.1. This step is semi-automatic, requiring the programmer to select the regions of the code that can be parallelised using high-level parallel patterns and to choose the appropriate hybrid pattern from the Skel library.

3.1 Lapedo Example in C++

We will use the parallelisation of an Image Convolution to illustrate how the LAPEDO framework can be used in practice for C++/Fastflow. Image convolution is a technique widely used in image processing applications such as blurring, smoothing and edge detection. The sequential structure (shown below), consists of two stages. The first stage, reads an image from a file, while the second stage, applies a filter to each image. The convolution process is typically applied to a stream of images. Computationally, the filtering stage requires a scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels:

\[
output\_pixel(i, j) = \sum_m \sum_n input\_pixel(i - n, j - m) \times filter\_weight(n, m) \quad (3.1)
\]

The original sequential C++ code is shown below.

```c++
int main(int argc, char * argv[]) {
    ...
    images = (char **) malloc(sizeof(char *)*NIMGS);
    for (int i=0; i<NIMGS; i++) {
        images[i] = (char *) malloc(sizeof(char)*1000);
        sprintf(images[i],"images/image0.png");
        void *res;
        res = create_input(images[i]);

    }

task_t *t = (task_t *)res;
int vstep = (maskWidth)/2;
int hstep = (maskHeight)/2;
float sumFX;
int left,right,top,bottom,maskIndex,index;
...
for(int x = 0; x < height; x++)
for(int y = 0; y < width; y++) {
    left = (x < vstep) ? 0 : (x - vstep);
    right = ((x + vstep - 1) >= width) ?
                width - 1 : (x + vstep - 1);
    top = (y < hstep) ? 0 : (y - hstep);
    bottom = ((y + hstep - 1) >= height) ?
                height - 1 : (y + hstep - 1);
    sumFX = 0;
    for(int i = left; i <= right; i++)
        for(int j = top; j <= bottom; j++) {
            maskIndex = (j - (y - hstep)) *
                         maskWidth + (i - (x - vstep));
            index = j * width + i;
            sumFX += ((float)t->inpt[index] *
                        t->msk[maskIndex]);
        }
    sumFX += 0.5f;
    t->outpt[y*width + x] = (ushort) sumFX;
}
}

1. The first step is to extract the body of code into a function that can be called
easier from a FastFlow ff_node. To do this, we simply select the code
in the body of the for loop (as above) and choose the Extract Function
refactoring (as described in BLAH). This refactoring prompts for the name
of the new function to be extracted. The result is that the highlight loop body
expression is replaced with a function call where the free variables in the
highlighted expression are passed as formal parameters to the function. The
result of extracting the functions is shown below.

```c
void * filter(task_t *t) {
    int vstep = (maskWidth)/2;
    int hstep = (maskHeight)/2;
    float sumFX;
```
int left, right, top, bottom, maskIndex, index;
for(int x = 0; x < height; x++)
    for(int y = 0; y < width; y++) {
        left = (x < vstep) ? 0 : (x - vstep);
        right = ((x + vstep - 1) >= width) ?
                width - 1 : (x + vstep - 1);
        top = (y < hstep) ? 0 : (y - hstep);
        bottom = ((y + hstep - 1) >= height) ?
                height - 1 : (y + hstep - 1);
        sumFX = 0;
        for(int i = left; i <= right; i++)
            for(int j = top; j <= bottom; j++) {
                maskIndex = (j - (y - hstep)) *
                             maskWidth + (i - (x - vstep));
                index = j * width + i;
                sumFX += ((float)t->inpt[index] *
                           t->msk[maskIndex]);
            }
        sumFX += 0.5f;
        t->outpt[y*width + x] = (ushort) sumFX;
    }
}
images = (char **) malloc (sizeof(char *)*NIMGS);
for (int i=0; i<NIMGS; i++) {
    images[i] = (char *) malloc (sizeof(char)*20);
    sprintf(images[i],"images/image%d.png", i);
}
for(int i = 0 ; i < NIMGS; i++)
{
    r1 = generate(images[i]);
    r2 = filter(r1);
}

2. The next step is to identify the components required as worker stages of the skeletons to be introduced. In order to do this, the programmer first selects the generate function call, and uses the Introduce Component refactoring. The result of this is a component declaration, genStage together with a call to the method callWorker, is shown below. The programmer repeats this step for the filter function call. The names for the components are given as a parameter to the refactoring tool before the refactoring is performed. The refactoring tool automatically checks for conflicts in scope.
Component<ff_task_t> genStage(generate);
Component<ff_task_t> filterStage(filter);

for (int i=0; i<NIMGS; i++) {
    r1 = genStage.callWorker(images[i]);
    results[i] = filterStage.callWorker(r1);
}

3. The next step is to add a Pipeline skeleton and identify that genStage and filterStage are the two stages of the pipeline. First, a new pipeline declaration, pipe, is introduced, using the Introduce Pipeline refactoring (from D4.2). The location of the new pipeline declaration is defined by the cursor position in the Eclipse IDE. In this example, the programmer chooses to introduce the new declaration just before the for loop. The next step is to add the two stages to the pipeline, by selecting the for loop, using the Introduce Pipeline Stages refactoring. The refactoring automatically checks the body of the for loop for Component instances, replacing calls to callWorker with pipe.addStage. In addition, the refactoring tool also automatically adds a preliminary streaming stage, streamgen, which acts as a stage that streams images to the genStage pipeline stage. This is determined as being the first argument to genStage before the refactoring took place. The refactored code is shown below.

```cpp
StreamGen streamgen(NIMGS, images);
ff_pipeline pipe;
pipe.add_stage(&streamgen);
pipe.add_stage(new genStage);
pipe.add_stage(new filterStage);
if (pipe.run_and_wait_end()<0) {
    error("running_pipeline\n");
    return -1;
}
```

4. The next step in the refactoring process is to farm the two pipeline stages. First, the programmer declares a new farm declaration, gen_farm (the name of the farm is programmer-defined) using the Introduce Farm refactoring.
(from D4.2). Then, the programmer selects the expression `new genStage` as the farm worker, and performs the Instantiate Farm refactoring. The refactoring tool then automatically adds the new worker to the `gen_farm` farm. The refactored code is shown below. Finally, the second stage of the pipeline is also farmed. Again, a new farm declaration is introduced, `filter_farm`, the expression `new filterStage` (second stage of the pipeline) is selected and the Instantiate Farm refactoring is performed. The result is shown below, where the `filter_farm` declaration adds the `filterStage` component as its worker. Additionally the farms are also added to the pipeline.

```cpp
ff_farm<> gen_farm;
global_farm.add_collector(NULL);
std::vector<ff_node*> gw;
for (int i=0; i<nworkers; i++)
    gw.push_back(new genStage);
global_farm.add_workers(gw);

ff_farm<> filter_farm;
global_farm2.add_collector(NULL);
std::vector<ff_node*> gw2;
for (int i=0; i<nworkers2; i++)
    gw2.push_back(new filterStage);
global_farm2.add_workers(gw2);

images = (char **) malloc (sizeof(char *)*NIMGS);
for (int i=0; i<NIMGS; i++) {
    images[i] = (char *) malloc (sizeof(char)*20);
    sprintf(images[i],"images/image%d.png", i);
}

StreamGen streamgen(NIMGS,images);

ff_pipeline pipe;
pipe.add_stage(&streamgen);
pipe.add_stage(&gen_farm);
pipe.add_stage(&filter_farm);

if (pipe.run_and_wait_end()<0) {
    error("running pipeline\n");
    return -1;
}
```

5. The next step requires the user to supply a pre-written OpenCL Kernel. The
LAPEDO framework then generates the OpenCL wrapping C++ logic, which is embedded into a FastFlow `ff_node`, using the heterogeneous FastFlow implementation from (D2.9). The generated code consists of three methods in a class overriding `ff_node`. These methods are `svc_SetUpOclObjects`, a method which is called when the class is created and sets up essential buffers and the kernel address space. `svc`, the main FastFlow method which is called when a task is sent to that FastFlow worker. This `svc` method marshals the task to the kernel, executes the kernel, and waits for the result to be sent back. `svc_releaseOclObjects`, which is called when the class is destructed, and frees up the buffers.

The generated code for each of these three methods is shown below.

```c++
void svc_SetUpOclObjects(cl_device_id dId)
{
    context = clCreateContext(NULL,1,&dId,NULL,NULL,&status);
    std::string kernelPath = "/SimpleConvolution_Kernels.cl";
    std::ifstream t(kernelPath.c_str());
    std::stringstream buffer;
    buffer << t.rdbuf();
    std::string source_string = buffer.str() + '\0';
    const char * source = source_string.c_str();
    size_t sizeSource[] = { strlen(source) };
    program = clCreateProgramWithSource(context,1,
        (const char **)&source,sizeSource,&status);
    if (status != CL_SUCCESS)
        printStatus("CreateProgramWithSource:", status);

    if ((status = clBuildProgram(program,1,&dId,NULL,NULL,NULL))
        != CL_SUCCESS)
        printStatus("CreateProgramWithSource", status);
    size_t len;
    char *buff;
    clGetProgramBuildInfo(program, dId, CL_PROGRAM_BUILD_LOG, 0,
        NULL, &len);
    buff = new char[len];
    clGetProgramBuildInfo(program, dId, CL_PROGRAM_BUILD_LOG, len,
        buff, NULL);

    cl_command_queue_properties prop = 0;
    commandQueue = clCreateCommandQueue(context, dId, prop, &status);

    inputBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY,
        sizeof(ushort) * width * height, NULL, &status);
    outputBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
        sizeof(ushort) * width * height, NULL, &status);
    maskBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY,
        sizeof(cl_float) * maskWidth * maskHeight, NULL, &status);
```

24
inputDimension = width;
maskDimension = maskWidth;

kernel = clCreateKernel(program, "simpleConvolution", &status);
if (status != CL_SUCCESS)
    printStatus("clCreateKernel", status);

status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
    (void *)&outputBuffer);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
    (void *)&inputBuffer);
status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
    (void *)&maskBuffer);
status = clSetKernelArg(kernel, 3, sizeof(cl_uint),
    (void *)&inputDimension);
status = clSetKernelArg(kernel, 4, sizeof(cl_uint),
    (void *)&maskDimension);

status = clGetKernelWorkGroupInfo(kernel, dId,
    CL_KERNEL_WORK_GROUP_SIZE,sizeof(size_t),
    &kernelWorkGroupSize,0);
}

void * svc(void * task) {
    if (task==NULL) { printf("null\n"); return NULL; }
    else{
        task_t *t =(task_t*)task;
        cl_int status;
        cl_event events[2];
        size_t globalThreads[1];
        size_t localThreads[1];

        globalThreads[0] = width*height;
        localThreads[0] = maskWidth*maskHeight;

        double gpu_start_time = get_current_time();
        status = clEnqueueWriteBuffer(commandQueue,inputBuffer,
            CL_FALSE,0,sizeof(ushort) * width * height, t->inpt,
            0,NULL,NULL);

        status = clEnqueueWriteBuffer(commandQueue,maskBuffer,
            CL_FALSE,0,sizeof(cl_float) * maskWidth * maskHeight,
            t->msk,0,NULL,NULL);
    }
if ((status = clEnqueueNDRangeKernel(
    commandQueue, kernel, 1, NULL, globalThreads,
    localThreads, 0, NULL, &events[0])) != CL_SUCCESS
    printStatus("enqueueNDRangeKernel", status);

    /* wait for the kernel call to finish execution */
    if ((status = clWaitForEvents(1, &events[0])) != CL_SUCCESS)
        printStatus("clWaitForEvents after enqueueNDRangeKernel", status);

    /* Enqueue readBuffer*/
    if ((status = clEnqueueReadBuffer(
        commandQueue, outputBuffer, CL_TRUE, 0,
        width * height * sizeof(ushort),
        t->outpt, 0, NULL, &events[1])) != CL_SUCCESS)
        printStatus("clEnqueueReadBuffer", status);
    /* Wait for the read buffer to finish execution */
    if ((status = clWaitForEvents(1, &events[1])) != CL_SUCCESS)
        printStatus("clWaitForEvents after clEnqueueReadBuffer", status);
    status = clReleaseEvent(events[0]);
    status = clReleaseEvent(events[1]);

    free(t->inpt);
    free(t->outpt);
    free(t->msk);
    free(t);

    return GO_ON;
}

void svc_releaseOclObjects(){
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(commandQueue);
    clReleaseMemObject(inputBuffer);
    clReleaseMemObject(maskBuffer);
    clReleaseMemObject(outputBuffer);
    clReleaseContext(context);
}

6. The final stage is to refactor our farm on the Filter stage into a hybrid farm,
so that it calls the GPU kernel instead.

Shown below
3.2 Lapedo Example in Erlang

We will use the parallelisation of an *N-Body Simulation* to illustrate how the LAPEDO framework can be used in practice for Erlang. This application is a simulation of a dynamic system of particles under the influence of external forces. It is commonly used in astronomy, for example, to simulate movements of the planets. The simulation proceeds in a number of iterations, each of which computes a new position for each particle, based on its interactions with all the other particles in the system. The core of the application is the *nbody* function:

\[
\text{nbody}(\text{Parts}, \text{Dt}, 0) \rightarrow \text{Parts}; \\
\text{nbody}(\text{Parts}, \text{Dt}, \text{NrIters}) \rightarrow \\
\quad \text{NewParts} = \text{lists:map}(\text{fun}(X) \rightarrow \\
\quad \quad \text{nbody\textunderscore\text{One}\textunderscore\text{part}\textunderscore\text{cpu}}(X, \text{Parts}, \text{Dt}) \text{ end}, \text{Parts}), \\
\quad \text{nbody}(\text{NewParts}, \text{Dt}, \text{NrIters}-1).
\]

This function applies the *nbody\textunderscore\text{One}\textunderscore\text{part}\textunderscore\text{cpu} function to each element of the input list of particles, *Part*. *Dt* is a constant that is passed to the *nbody\textunderscore\text{One}\textunderscore\text{part}\textunderscore\text{cpu} function. The *nbody\textunderscore\text{One}\textunderscore\text{part}\textunderscore\text{cpu} function is given below:

\[
\text{nbody\textunderscore\text{One}\textunderscore\text{part}\textunderscore\text{cpu}}(\text{Part}, \text{Particles}, \text{Dt}) \rightarrow \\
\quad \{\text{Ax}, \text{Ay}, \text{Az}\} = \text{calc\textunderscore\text{acc\textunderscore}\text{vector\textunderscore}\text{list}}(\text{Part}, \text{Particles}), \\
\quad \{\text{X}, \text{Y}, \text{Z}, \text{M}, \text{Vx}, \text{Vy}, \text{Vz}, _\} = \text{Part}, \\
\quad \text{Xnew} = \text{X} + \text{Dt}\text{\ast}\text{Vx} + 0.5\text{\ast}\text{Dt}\text{\ast}\text{Ax}, \\
\quad \text{Ynew} = \text{Y} + \text{Dt}\text{\ast}\text{Vy} + 0.5\text{\ast}\text{Dt}\text{\ast}\text{Ay},
\]
\[
\begin{align*}
Z_{\text{new}} &= Z + Dt \cdot V_{z} + 0.5 \cdot Dt \cdot Dt \cdot A_{z}, \\
V_{x\text{new}} &= V_{x} + Dt \cdot A_{x}, \\
V_{y\text{new}} &= V_{y} + Dt \cdot A_{y}, \\
V_{z\text{new}} &= V_{z} + Dt \cdot A_{z}, \\
\{X_{\text{new}}, Y_{\text{new}}, Z_{\text{new}}, M, V_{x\text{new}}, V_{y\text{new}}, V_{z\text{new}}, 0\}. 
\end{align*}
\]

The auxiliary \texttt{calc\_acc\_vector\_list} function is a simple fold over the list of particles:

\[
\text{calc\_acc\_vector\_list } (\text{Part,Particles}) \rightarrow \\
\text{lists:foldl(fun(X,Sum) } \rightarrow \text{ add\_to\_acc\_list(X,Sum,Part)} \text{ end,} \\
\{0,0,0\}, \text{ Particles}).
\]

We omit the definition of the \texttt{add\_to\_acc\_list} function. We also assume that we have available an OpenCL kernel \texttt{nbodyKernel} that applies the \texttt{nbody\_one\_part\_cpu} function to a chunk of particles. We can see that the main part of the code (in the \texttt{nbody} function) involves applying the same function to a list of particles. This suggests that it can be parallelised by using a \texttt{map} or \texttt{farm} skeleton. Since all of the inputs are available in advance of the computation, each individual computation on particles is relatively small but the total number of particles involved is potentially large. We have, therefore, chosen to use a \texttt{map} pattern, and use Lapedo for parallelisation as follows:

1. We first create an initial parallelisation by introducing the \texttt{map with chunking} skeleton in the body of the \texttt{nbody} function using the \texttt{ParMapIntro} refactoring. In order to do this, we first need to shape the code appropriately. In particular, since we want our map worker function to apply \texttt{nbody\_one\_part\_cpu} function to a chunk of input particles, rather than just one particle. We, therefore, produce a new function, \texttt{nbody\_chunk} that does exactly this. For that, we use \texttt{ListChunking} refactoring. This results in the following parallel CPU-only code:

\[
\begin{align*}
n\text{body\_chunk}(\text{Chunk, Parts, Dt}) \rightarrow \\
\text{List:foldl(fun(X,Sum) } \rightarrow \text{ add\_to\_acc\_list(X,Sum,Part)} \text{ end,} \\
\{0,0,0\}, \text{ Particles}).
\end{align*}
\]

\[
\begin{align*}
n\text{body}(\text{Parts, Dt, NrIters,NrWorkers}) \rightarrow \text{Parts}; \\
n\text{body}(\text{Parts, Dt, NrIters,NrWorkers}) \rightarrow \\
\text{Map } = \text{\{map, (seq, fun(X) } \rightarrow \text{nbody\_chunk(X, Parts, Dt)} \text{ end\},} \\
\text{f(X) } \rightarrow \text{chunk(X, NrWorkers),} \\
\text{f(X) } \rightarrow \text{X\}}, \\
\text{NewParts } = \text{skel:do(\{Map\},\{Parts\}),} \\
n\text{body}(\text{NewParts, Dt, NrIters-1}).
\end{align*}
\]
2. We then need to perform some more program shaping transformations in order to prepare the code for the introduction of the hybrid map with chunking pattern and/or to improve on the initial CPU-only parallelisation. Our GPU kernel already operates over chunks of particles, similar to the \texttt{nbody\_chunk} function. Therefore, for the introduction of the hybrid map with chunking pattern, we only need to modify the \texttt{nbody\_chunk} function to work on binary data, instead of lists. We, therefore, apply the \textit{ListToBinary} program shaping refactoring. Since Erlang binaries are essentially just streams of bytes, we need to specify the size (in bytes) of each data item (particle in our case), so that we can correctly apply the binary map and fold functions. The resulting code introduces the \texttt{nbody\_chunk\_binary} function in place of the \texttt{nbody\_chunk} function from the previous step.

\begin{verbatim}
\texttt{nbody\_chunk\_binary(Chunk, Particles,Dt) ->
  binary8Map(fun nbody\_one\_part\_cpu/3, Chunk).}
\end{verbatim}

The \texttt{binary8Map} function maps over a binary object where each data item is precisely 8 bytes long. Since it contains no explicit operations over binaries, the body of the \texttt{nbody\_one\_part\_cpu} function remains unchanged. However, the \texttt{calc\_acc\_vector\_list} must be changed to use a binary fold rather than the original list version:

\begin{verbatim}
\texttt{calc\_acc\_vector\_list (Part,Particles) ->
  binary8Foldl(fun(X,Sum) ->
    add\_to\_acc\_list(X,Sum,Part) end,
    \{0,0,0\}, Particles).}
\end{verbatim}

As with \texttt{binary8Map}, \texttt{binary8Foldl} operates over 8-byte wide binary data.

In addition, we can observe that the \texttt{nbody\_chunk} operates over lists of particles. Since this is a worker function of the map with chunking skeleton, the lists of particles will have to be transferred to workers. This may be quite expensive, depending on the number of particles. Binary data structure avoids these costs by passing pointers to binaries to the workers, instead of whole binaries, but the access to binaries is quite slow. So, we may want to improve the CPU-only parallelisation by using ETS tables for chunks of
works and the particles list. We can do this with \textit{ListToEts} refactorings, rewriting the body of the \texttt{nbody} function into:

\begin{verbatim}
  nbody(Parts, _Dt, 0, _NrWorkers) ->
      Parts;
  nbody(Parts, Dt, NrIters, NrWorkers) ->
      table = create_ets_table(),
      true = ets:insert_new(table, 0, Particles),
      Tasks = lists:seq(1, NrWorkers),
      Map = {map, {seq, fun(X) ->
             nbody_chunk_ets(X, Dt, table)
          end},
             f(X) -> chunk_ets(X, NrWorkers),
             f(X) -> X},
      NewParts = skel:do([Map], [Tasks]),
      nbody(NewParts, Dt, NrIters-1, NrWorkers).
\end{verbatim}

We omit the \texttt{chunk_ets} function. The specialised \texttt{nbody_chunk_ets} function is given below:

\begin{verbatim}
  nbody_chunk_ets(Index, Dt, table) ->
      [[0, Particles]] = ets:lookup(table, 0),
      [[I, Chunk]] = ets:lookup(table, Index),
      R = lists:map (fun(X) ->
                     nbody_one_part_cpu (X, Particles, Dt) end, Chunk),
      ets:insert(table, [I, R]),
      I.
\end{verbatim}

Note that the worker function, \texttt{nbody_chunk_ets} assumes that the particles, as well as all chunks of work, are stored in the shared ETS table \texttt{table}. The entries in the table are hashed by the index, with index 0 mapping to the list of particles, and indices from 1 to \texttt{NrWorkers} mapping to the chunks of work to be assigned to the workers. The worker function then simply reads the appropriate index, passed to it as a parameter, fetches its chunk of work and the list of particles and applies the \texttt{nbody_one_part_cpu} function to each of the particles. Afterwards, it overwrites the entry in the ETS table with the appropriate index with the updated chunk of particles, so that it can be used in the next iteration of the algorithm.

3. We can then invoke the GPU Code Generator to generate a wrapper around the GPU kernel, creating a GPU component, \texttt{nbody_chunk_gpu}, whose
functionality is identical to that of nbody_chunk_binary. Certain parameters, such as `K#kwork.localSize` and `K#kwork.globalSize`, cannot be automatically generated at present, however, and therefore need to be supplied by the programmer. The generated code is shown below:

```erlang
nbody_chunk_gpu(Chunk, Particles, Dt) ->
    %% set up the environment
    E = clu:setup(all),
    {ok,Program} = clu:build_source(E, program(ok)),
    {ok,Kernel} = cl:create_kernel(Program, "nbodyKernel"),
    ...

    %% create buffers
    {ok,ChunkInBuffer} = cl:create_buffer(E#cl.context, [read_only], byte_size(Chunk)),
    {ok,ParticlesBuffer} = cl:create_buffer(E#cl.context, [read_only], byte_size(Particles)),
    {ok,ChunkOutBuffer} = cl:create_buffer(E#cl.context, [read_only], byte_size(Chunk)),

    ...

    %% data transfers
    {ok,E1} = cl:enqueue_write_buffer(K#kwork.queue, ChunkInBuffer, 0, byte_size(Chunk), Chunk, []),
    {ok,E2} = cl:enqueue_write_buffer(K#kwork.queue, ParticlesInBuffer, 0, byte_size(Particles), Particles, []),

    ...

    %% Set kernel arguments
    ok = cl:set_kernel_arg(Kernel, 0, ChunkInBuffer),
    ok = cl:set_kernel_arg(Kernel, 1, ParticlesInBuffer),

    ...

    %% enqueue the kernel
    Global = K#kwork.globalSize,
    Local = K#kwork.localSize,
```
{ok, E6} = cl:enqueue_nd_range_kernel(K#kwork.queue, Kernel, [Global], [Local], [E1, E2]),

...

%% read back from the GPU
{ok, E7} = cl:enqueue_read_buffer(K#kwork.queue, ChunkOutBuffer, 0, byte_size(Chunk), [E6]),
Result = case cl:wait(E7, 3000) of
  {ok, Data} ->
    Data;
  Res3 ->
    Res3
end,

...

%% return the result
Result.

4. Finally, since we now have the CPU and GPU components (nbody_chunk_binary and nbody_chunk_gpu), and both operate over binary chunks, we can introduce the map with chunking pattern. We apply the IntroduceHybridMap refactoring from Section 3.5.1 to obtain the following hybrid version of the nbody function.

nbody(Parts, _Dt, 0, _NCPUWorkers, _NGPUWorkers) ->
Parts;
nbody(Parts, Dt, NrIters, NCPUWorkers, NGPUWorkers) ->
HybMap =
  hyb_map(NCPUWorkers, NGPUWorkers, fun(X) -> nbody_chunk_binary(X, Parts, Dt) end, fun(X) -> nbody_chunk_gpu(X, Parts, Dt) end, splitFun),
NewParticles = skel:do([HybMap], [Particles]),
nbody(NewParticles, Dt, NrIters-1, NCPUWorkers, NGPUWorkers).

For more details about hybrid map with chunking, see Section 3.4. Note that we need to provide the splitFun function that splits the list of particles into
chunks that are sent to individual workers. This function, that is generated automatically, can split the list into chunks of even or uneven size, depending on the performance of CPU and GPU workers. For more details on splitting the work between CPUs and GPUs, see 3.4.3.

3.3 GPU Programming

3.3.1 GPU Programming in Erlang

Erlang has no native support for GPU programming. However, a library containing OpenCL bindings is available\(^1\), This provides an Erlang interface to low-level OpenCL functions to set up GPU computations, transfer data to/from the CPU, and launch GPU kernels implemented in OpenCL plus basic marshalling mechanisms between binary data structures in Erlang and C arrays. While enabling programmers to write their code in Erlang, this library does not simplify GPU programming, since the programmer is still required to write code that is equivalent to programming directly in OpenCL. In the LAPEDO framework, we build on this library and automate the process of generating the bookkeeping OpenCL code.

3.4 Hybrid Skeletons in Lapedo

In LAPEDO, we use hybrid implementations of various generic skeletons (the skeletons themselves are described in D2.1).

3.4.1 Hybrid Skeletons in C++/Fastflow

For C++, we make use of the hybrid FastFlow skeletons implemented and described in Deliverable D2.4.

3.4.2 Hybrid Skeletons in Erlang

Hybrid Map. The hybrid map skeleton, hyb_map, requires two implementations of the worker function, one for CPU and another for GPU workers. The two implementations need to have the same interface and produce the same result. Hybrid map also requires the number of each type of worker to use. Internally, hyb_map is implemented in terms of map skeleton, but with specialised worker and splitting functions:

\[
\text{hyb\_map} = \text{fun}(\text{NrCPUWs, NrGPUWs, CPU\_W, GPU\_W, Split}) \rightarrow
\{
\text{map, } [[\text{seq, fun(T)} \rightarrow
\text{hyb\_map:hyb\_dispatcher (CPU\_W, GPU\_W, T)} \text{ end}],
\text{fun(X)) \rightarrow hyb\_map:hyb\_split(X, Split, NrCPUWs, NrGPUWs)},
\}
\]

\(^1\)Available to download at https://github.com/tonyrog/cl
fun combine/1} end.

hyb_map:hyb_split takes the splitting function, Split, applies it to the input list (returning a list of tasks as a result) and then tags each task with either cpu or gpu tag.

hyb_split(InputList, Split, NrCPUWs, NrGPUWs) ->
TaskList = Split(InputList),
TaggedTaskList = tag_list(TaskList, NrCPUWs, NrGPUWs),
TaggedTaskList.

hyb_map:hyb_dispatcher then simply reads the tag of a task and, depending on it, executes CPU or GPU worker function. In the case of the GPU worker function, first the OpenCL environment is set, then buffers are created on a GPU, the required data is transferred, the user-supplied GPU kernel is executed, the result is fetched back and returned.
hyb_dispatcher =
fun(CPUWorkerFunction, GPUWorkerFunction, Task) ->
case Task of
  {cpu, InputData} -> CPUWorkerFunction(InputData);
  {gpu, InputData} -> GPUWorkerFunction(InputData)
end.

Section 3.5.3 describes a refactoring for automatically generating all the GPU code, and Section 3.4.3 describes methods for dividing work between CPUs and GPUs that can be used to automatically provide the Split function for hyb_map skeleton.

Hybrid Farm. The hybrid task farm skeleton, hyb_farm, is similar to hyb_map, except that it does not require a splitting function: we assume that its input is a stream of already constructed tasks.

hyb_farm = fun(NrCPUWs, NrGPUWs, CPU_W, GPU_W) ->
  {farm, [{seq, fun(X) ->
           hyb_farm:hyb_dispatcher(CPU_W, GPU_W, X)
           end}], NrCPUWs+NrGPUWs}.

Before calling the hyb_farm skeleton, we need to tag the tasks in the input stream, similar to with hyb_map:

runFarm = fun(NCPUWs,NGPUWs) ->
 ListOfTasks = <construct a list of tasks>,
TaggedList = tag_list (ListOfTasks, NCPUWs, NGPUWs),
skel:do {hyb_farm(NCPUWs, NGPUWs, fun cpu_w/1, fun gpu_w/1),
          TaggedList}.

As with hyb_map, hyb_farm:hyb_dispatcher will schedule tasks as appropriate on the available CPU/GPU workers.

3.4.3 Division of Work Between CPUs and GPUs.

The hyb_map and hyb_farm skeletons require the numbers of CPU and GPU workers to be specified explicitly (NrCPUWs and NrGPUWs). In the cases where there is no nesting of skeletons, i.e. where there is only a hyb_map or hyb_farm skeleton at the top level, and the inner skeleton for it is func, we can simply
set the number of CPU workers to be equal to the number of cores in a CPU, and the number of GPU workers to be the number of physical GPU devices in the system. The problem with this, however, is that for suitable problems, GPUs are much faster in processing tasks than CPU cores. map and farm skeletons (and, consequently, hyb_map and hyb_farm) use push approach, where tasks are pushed in round-robin fashion to the workers. Therefore, in the case of farm and hyb_farm skeletons, the same amount of work is assigned to each worker. This is also the case for map and hyb_map skeletons, if we use a simple splitting function that divides an input list into equally-sized chunks. If GPUs can process tasks much faster, equal division of work is obviously a problem, since GPUs will finish much faster than CPU cores and will, therefore, have to wait until CPU cores finish their portion of work.

In the case of map, the aforementioned problem can be avoided if we do a smarter division of input lists into chunks. Assuming that a given problem is regular (i.e. that it takes the same amount of time to process each element of an input list) and that we can obtain timing information (e.g. using profiling) that can tell us how faster can a GPU process one list item (or a set of list items) than a CPU core, we can, using some simple formulas, derive how many list items should be processed by the GPUs and how much by each CPU core in order to get the best execution time. For example, assume that we have $g$ GPU and $c$ CPU cores in a system, and that the ratio between processing time for $k$ items between a CPU and a GPU is $\text{ratio}$. If an input list has $n$ items (where $n$ is divisible by $k$), then we can estimate the time it takes to process all of the items in the list if $n_c$ items are processed by CPU cores by

$$T(n_c) = \max \left\{ \left\lceil \frac{nC}{c} \cdot \text{ratio} \right\rceil , \left\lceil \frac{n-n_c}{g} \right\rceil \right\},$$

where the first argument of the max is the time it takes to process $nC$ items on CPU cores, and the second argument is the time it takes to process the remaining items by the GPUs. The best time we can obtain is then $\min \{T(nC) | nC \in \{0, k, 2k, ..., n\} \}$, and the optimal number of items to process on CPU cores is such $nC$ for which this minimum is obtained. In this way, we calculate a pair $(nC, n - nC)$, the number of list items to be processed by CPU cores and the GPUs, respectively. Chunk sizes for CPU cores are then

$$\left\{ \left\lfloor \frac{nC}{c} \right\rfloor , \left\lfloor \frac{nC}{c} \right\rfloor , \ldots , \left\lfloor \frac{nC}{c} \right\rfloor , \left\lceil \frac{nC}{c} \right\rceil , \left\lceil \frac{nC}{c} \right\rceil , \ldots , \left\lceil \frac{nC}{c} \right\rceil \right\}_{c-(n_c \mod c \times \text{times}}^{n_c \mod c \times \text{times}}.$$

We can similarly calculate the chunk sizes for the GPUs. The parameter $k$ in the above discussion should be chosen so that it gives the best parallelism on the GPU, i.e. it should be maximum number of list items that the GPU can process in parallel (see Chapter 3.7 for examples).
HybMapIntroSeq(\(\rho, e, g, n_{CPU}, n_{GPU}, s, com\)) =
\[\mathcal{E}[e] \Rightarrow [[\text{map}, [[\text{seq}, \text{fun}(x) \rightarrow \\
\text{het_map} : \text{het_dispatcher}(\text{lists} : \text{map}(\text{fun}(y) \rightarrow c', y), g, x) \text{end})), \\
\text{fun}(x) \rightarrow \text{het_map} : \text{het_split}(s, x, n_{CPU}, n_{GPU}) \text{end, com}]]
\]
\{seq \in \rho, map \in \rho, g \in \rho, x \text{ fresh}, y \text{ fresh}, com \in \rho, s \in \rho, c \in \rho, \\
\text{het_map} \in \text{imports}(\rho), \text{het_dispatcher} \in \rho, \text{het_split} \in \rho, \text{free}(c, \rho)\}

where \(c' = [[\text{fun}(\arg \mathcal{E}) \rightarrow c]], e = [[\text{seq}, c]]\)

Figure 3.2: Refactoring rules for the Introduce Hybrid Map refactoring

For \text{farm} skeleton, we can, if we know the length of the stream of tasks, calculate how many tasks should be evaluated by CPU cores and how much by the GPUs using a similar approach to above. Additionally, \text{farm} can be implemented using the pull (work stealing \cite{10}) approach, so that the tasks are sent to the workers on-demand when they request them, as opposed to eagerly in the push approach. However, this method requires more communication, as the workers need to send messages to the emitter to request more work. It is, however, the preferred solution when tasks are irregular (i.e. when some tasks are more computationally-intensive than others). Since the examples that we consider in this paper are regular, we do not use the work-stealing approach.

In the case where there is a nesting of skeletons (e.g. \text{farm} inside of a \text{map}), we cannot simply let all \text{farms} use all CPU cores and GPUs, since that would result in overloading the system with processes. For complex nestings of skeletons, it is highly non-trivial to derive the “correct” number of CPU and GPU workers for all \text{farms/maps}. In \cite{13}, we have described one method for that, based on Monte-Carlo Tree Search (MCTS) method. We plan to integrate this into \text{LAPEDO} in the future.

3.5 \text{LAPEDO} Refactorings for Hybrid Skeletons and Program Shaping

This section describes the new \text{LAPEDO} refactorings for introducing hybrid skeletons and for program shaping. Each refactoring is described in terms of the transformation rules and pre-conditions, using the refactoring rules from D4.1.

3.5.1 Refactorings to Introduce Hybrid Skeletons

The Introduce Hybrid Map refactoring is used to insert an instance of a hybrid map (Figure 3.2). \text{HybMapIntroSeq} requires an environment, \(\rho\); a highlighted expression, \(e\); a function, \(g\), that has been generated using the GPU offloading generator from Section 3.5.3; the number of CPU workers, \(n_{CPU}\); the number of GPU workers, \(n_{GPU}\); a function, \(s\), that is used to \text{split} the data, and a function, \(com\), that is
ListToBinary(\(\rho, f, \text{Arg}, T, \rightarrow\rightarrow\) types) =

\[ D[f] \Rightarrow \{ f(\overrightarrow{x}, \text{Arg}, \overrightarrow{z}) \rightarrow \text{body}' \} \]
\{ f \in \rho, \overrightarrow{x} \in \rho, \overrightarrow{z} \in \rho, \text{Arg} \in \rho, T \notin \rho \}

where

\[ \text{Arg}' = \text{ListToBinaryExpr}(\rho, \text{Arg}, T, \rightarrow\rightarrow\) types) \]
\[ \text{body}' = \text{ListToBinaryExpr}(\rho, \text{body}, T, \rightarrow\rightarrow\) types) \]
\[ f = [ f(\overrightarrow{x}, \text{Arg}, \overrightarrow{z}) \rightarrow \text{body}' ] \]

Figure 3.3: Top-level rewrite rule for the Transform List to Binary refactoring

ListToBinary takes as its arguments: an environment, \(\rho\); the function to be operated over, \(f\); the list, \(\text{Arg}\); an unbound variable name \(T\); and \(\rightarrow\rightarrow\) types, which indicates the type of individual binary elements. It first transforms \(\text{Arg}\), then the body of \(f\).

To illustrate this refactoring, consider the \text{get_particles} function:

---

**3.5.2 Refactorings for Program Shaping**

Program shaping refactorings do not introduce any parallelism themselves, but prepare the original source so that it is easier to introduce parallelism, for example, by altering data representations. These refactorings are useful for improving memory and space usage, and are also required for marshalling data to a GPU.

**3.5.2.1 List to Binary.**

The Erlang concurrency model is shared-nothing. This means that all data must be copied between processes. The two exceptions are binaries and ETS tables, which are normally passed by reference. Converting a shared list to a binary form can therefore have a significant positive impact on performance. Conversion is also necessary if a list needs to be passed to a GPU kernel. Top-level rewrite rules for the Transform List to Binary refactoring are shown in Figure 3.3. Full rules can be found in Appendix A.
get_particles(Data) ->
    lists:foldl(fun(X, Acc) ->
        {Str, _} = string:to_float(X),
        [Str | Acc] end,
    [], string:tokens(Data, " ")).

This is part of our N-Body example from Section 3.7.2. It takes a string, Data, that contains space-separated floating-point numbers, and returns a list of these numbers. Suppose that, instead of a list, we require get_particles to return a binary. We can achieve this by applying ListToBinary to the body of get_particles:

get_particles(Data) ->
    lists:foldl(fun(X, Acc) ->
        {Str, _} = string:to_float(X),
        T = <<Str/float>>, <<T/binary, Acc/binary>> end,
    <<>>, string:tokens(Data, " ")).

3.5.2.2 List/Binary to ETS Table.
Erlang’s ETS tables provide a single shared, updatable structure, that is designed to store large amounts of data. ETS tables can be updated much more easily and quickly than binaries, without the need to create a new data structure. However, they are not thread safe: multiple processes can simultaneously modify the same part of the table. ETS tables should therefore only be used when it is possible to ensure the consistency of reads and updates. We can define a Transform List or Binary to ETS Table that is similar to ListToBinary.

3.5.3 Generating GPU Worker Code (Offloading)
We now describe our automatic tool for generating GPU worker code. A GPU worker is essentially a wrapper for a GPU kernel that performs the actual useful work, and is executed on a CPU. The wrapper takes care of data transfers between the CPU and GPU, sets appropriate parameters of the GPU kernel, and finally schedules the kernel for execution on a GPU. For now, we will assume that the kernel is provided by the programmer, although in future we intend to extend our approach to also automatically generate OpenCL kernels from (simplified) Erlang source code. The programmer starts by supplying an Erlang module that contains all the parameters that are needed for the generation process. This includes: the name of the Erlang module to be generated; its path; details of the GPU kernel arguments, including the names of the arguments, whether or not they are inputs or outputs to the kernel, and their size, in bytes; and finally a flag to indicate whether the parameter is local, global or private.
Generating the kwork Record. A new Erlang module with the specified name is now generated by the tool. The module includes the definition of a record, kwork, whose fields contain all the necessary kernel buffers, work group sizes, clock frequency settings etc., that are needed by the GPU kernel. Here, e1, e2 and e3 are generated names that refer to the events that will capture the enqueueing of data into the three programmer-supplied buffers.

```
-record(kwork, { program, kernel, queue, local, freq, units, weight, e1, e2, e3, imem, omem, isize, idata, chunkInBuffer, particlesInBuffer, chunkOutBuffer, result }).
```

Generating the Erlang Function. A new Erlang function is then generated, whose arguments are the supplied global and local sizes for the kernel, plus the data for the input buffers to the kernel. The first step in setting up the kernel involves loading and building the kernel. When this is done, the next step is to set various kernel parameters, such as the clock frequency, etc.:

```
do_nbody_gpu(Chunk, Particles, Dt, ChunkSize) ->
    E = clu:setup(all),
    {ok,Program} = clu:build_source(E, program(ok)),
    {ok,Kernel} = cl:create_kernel(Program, "nbody_kern"),
    Kws = map( fun(Device) ->
    {ok,Queue} = cl:create_queue(E#cl.context,Device,[]),
    ...
    #kwork{ queue=Queue, ...} end, E#cl.devices),
```

The next step involves creating the buffers that are needed to marshal the data that is transferred to and from the GPU kernel. This code is generated automatically using the supplied kernel arguments.
Kws3 = map( fun(K) ->
    {ok,ChunkInBuffer} = cl:create_buffer(E#cl.context,[read_only],byte_size(Chunk)),
    {ok,ParticlesBuffer} = ...
    K#kwork {chunkInBuffer=ChunkInBuffer, ... } end, Kws),

The next step involves enqueueing the data into the buffers, and then setting the kernel arguments, so that each buffer is assigned to the correct kernel argument. Once the data is enqueued, the kernel can be run on the GPU.

Kws4 = map( fun(K) ->
    {ok,E1} = cl:enqueue_write_buffer(K#kwork.queue, K#kwork.chunkInBuffer, 0, byte_size(Chunk), Chunk, []),
    {ok,E2} = ...
    ok = cl:set_kernel_arg(K#kwork.kernel, 0, K#kwork.chunkInBuffer),
    ...
    {ok,E6} = cl:enqueue_nd_range_kernel(K#kwork.queue, K#kwork.kernel, [Global], [Local], [E1,...]),
    {ok,E7} = cl:enqueue_read_buffer(K#kwork.queue, K#kwork.chunkOutBuffer, 0, byte_size(Chunk), [E6]),
    ok = cl:flush(K#kwork.queue),
    Result = cl:wait(E7)
    K2#kwork {result = Result, e1=E1,... }

Finally, the buffers are released and deallocated, and the result is returned.

Bs = map( fun(K) ->
    cl:release_mem_object(K#kwork.particlesInBuffer), ...
    end, Kws4),
    clu:teardown(E),
    Bs.

3.6 Applying LAPEDO to C++

LAPEDO is an in-principle proof of concept, and, as such, its principles and tech-
niques can easily be transferred and extended to other languages as well as Erlang. To demonstrate how it may be possible to apply similar techniques to C++, we have prepared an exemplar of image convolution. Here, the user provides a kernel (as in LAPEDO), and we envisage the offloading logic can be generated in much the same way as in LAPEDO, with similar refactorings to introduce the GPU component into FastFlow. The example is available at \(^2\).

3.7 Heterogeneous Refactoring Examples

In this section we evaluate our LAPEDO framework for both FastFlow/C++ and Erlang applications. For each application, we went through several phases in the process of parallelisation. Starting from an initial version that has no parallelism and a user-provided OpenCL GPU kernel: i) we first do an initial parallelisation by introducing CPU-only skeletons; ii) we refine this parallelisation using program shaping refactorings where applicable, in order to improve performance and/or to prepare the code for introducing hybrid skeletons; and, iii) we extend the initial parallelisation using hybrid skeletons. We omit the step where we generate C++/Erlang GPU code from a given OpenCL kernel, as the resulting code is very similar for all of the examples. All of the steps are carried out semi-automatically using the Eclipse Fastflow/C++ refactoring tool from D4.2 and PaRTE (from Chapter 4) and the refactorings described in Section 3.5 and Appendix A. We also evaluate the speedups that we obtain for various versions of the parallel code. We have used an experimental platform (called titanic, located at the University of Pisa), which has two 2.3GHz 12-core AMD Opteron 6176 processors, and an NVidia Tesla C2050 Fermi GPU with 448 CUDA cores, running CentOS Linux.

3.7.1 Examples in Fastflow/C++

For Fastflow/C++, we consider applications that belong to different domains, showing the generality of our parallelisation methodology. The applications we consider are Image Convolution, Ant Colony Optimisation and Molecular Dynamics. The evaluations of the skeleton configurations in Step 4 are performed on a machine comprising 2x2.4Ghz 12-core AMD Opteron 6176 CPUs, coupled with an NVidia Tesla C2050 graphic card with 448 CUDA cores running at 1.16GHz, running CentOS Linux and g++ 4.1.2. The speedups reported in the figures are averages over 5 independent runs. The results are reported in more detail in D3.2. We omit such detail here for brevity and report basic performance results only.

Image Convolution As described in Section 3.1, Image Convolution consists of two stages: a generation stage, which reads an image and processes it into an intermediary form; and, a filter stage, which applies a convolution filter to each image generated by the first stage. In our example from Section 3.1, we first introduce a

\(^2\)http://chrisb.host.cs.st-andrews.ac.uk/farm_mix_cpu_gpu.cpp
pipeline and then farm each stage of the pipeline. The farm in second stage of the pipeline in the transformed into a GPU computation. Figure 3.4 shows the results of this refactoring process. In the figure, $\Delta(g)$ corresponds to a farm of genStage functions; $\parallel$ refers to a pipeline, and $\Delta(f)$ corresponds to the filter stage on the GPU. The $x$ axis shows the number of CPU workers for $\Delta(g)$, whereas each line on the graph corresponds to a fixed number of GPU workers in $\Delta(f)$, with the number of CPU workers in $\Delta(f)$ being 0; this corresponds to the best speedups obtained for this configuration.

Ant Colony Optimisation Ant Colony Optimisation (ACO) [4] is a heuristic for solving NP-complete optimisation problems, inspired by the behaviour of real ants. In this paper, 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, $p_i$, deadline, $d_i$, and weight, $w_i$. The goal is to schedule the execution of jobs in a way that achieves minimal total weighted tardiness, where the tardiness of a job is defined by $T_i = \max\{0, C_i - d_i\}$ (with $C_i$ being the completion time of the job, $i$) and the total tardiness of the schedule is defined as $\sum w_i T_i$. 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. The pheromone trail is stronger along previously successful routes and is defined by a matrix, $\tau$, where $\tau[i, j]$ is the preference of assigning job $j$ to the $i$-th place in the schedule. After all
ants compute their solution, the best solution is chosen as the ‘running best’; the pheromone trail is updated accordingly, and the next iteration is started.

Figure 3.5 shows speedups for the $\Delta(s); p; u$ configuration. The basic structure of one iteration of the algorithm is $s; p; u$, where $s$ is the phase which finds the solutions for all ants, $p$ the phase which picks up the best solution and $u$ the phase where the pheromone trail is updated, taking into account the current best solution. Sequential ordering of the phases prevents introducing a pipeline between any two stages. Each line shows the speedups with a fixed number of GPU workers and varying number of CPU workers for $\Delta(s)$. From the figure, we can observe that the best speedup of 7.04 is obtained with (7, 5) CPU and GPU workers.

**Molecular Dynamics** Molecular Dynamics (MD) (supplied by HLRS) simulation computes a system of N particles on the atomic level [12]. Once the system is initialised, the interactions between the molecules are evaluated explicitly, allowing for the numerical integration of Newton’s equations of motion. The molecular trajectories in time yield the thermodynamic properties of the system.

The molecular simulation code used here (CMD) is designed for basic research into HPC MD. In the BasicN2 variant investigated in this paper, all intermolecular distances are evaluated in order to identify interaction partners. However, a special flavour of BasicN2 is used, where the domain is decomposed into subdomains of approximately 1000 molecules in order to counter the prohibitive scaling of neighbour search (otherwise $O(N^2)$). These subdomains are distributed among FastFlow CPU and GPU workers. As inferred from profiling data, the force calculation rou-
tine dominates the simulation time and is therefore parallelised. The force calculation itself is decomposed into two kernels, intra-domain and inter-domain (with the use of halos) interactions.

Figure 3.6 shows the speedups for a domain of 1000 molecules for the $\Delta(r \circ h)$ skeleton configuration, where $r$ denotes intra-domain interactions, and $h$ denotes inter-domain. In the figure, the $x$ axis corresponds to the number of CPU workers, and each line in the graph corresponds to a fixed number of GPU workers. In the figure, the best obtained speedup for this configuration is 23.43 for 22 CPU workers and 4 GPU workers.

3.7.2 Examples in Erlang

In this section, we evaluate our LAPEDO from Chapter 3 framework on four small but realistic Erlang examples, taken from different application domains: particle tracing (N-Body), evolutionary computing (Ant Colony Optimisation), image processing (Image Merge) and simulations (Football Simulation, from our Industrial Partner, Erlang Solutions).

**N-Body** The N-Body application was described in Chapter 3. As a reminder, the core of the application is the nbody function:

\[
\text{nbody}(\text{Parts}, 0) \rightarrow \text{Parts}; \\
\text{nbody}(\text{Parts}, \text{NrIters}) \rightarrow 
\]
The code is parallelised using the \texttt{map} (for the CPU-only versions) and \texttt{hyb_map} skeletons (for the CPU/GPU version). We have three different CPU-only versions: list version, binary version and ETS version. In the list version, particles, as well as chunks of works that need to be sent to the worker processes are kept in lists. Thus, when sending work to a worker process, the whole list needs to be copied, which can affect the performance. In the binary version, particles and chunks of work are kept in binaries, allowing us to avoid the costs of copying the data to the worker processes, but slowing down access to individual particles (as accessing elements of a binary is less efficient than accessing elements of a list). Finally, in the ETS version, particles and chunks of work are kept in ETS tables, allowing efficient access to individual particles (using hash tables) and avoiding the costs of copying the data.

Figure 3.7(a) shows the speedups that we obtained for the CPU-only versions of the code. The input consists of 20 000 particles, with randomly chosen initial coordinates. We can observe that all the versions scale reasonably well, and that there is a small, but notable benefit in using binaries over lists, and ETS tables over binaries. In this example, the amount of data that needs to be communicated to the worker processes is not very big (8 floating point numbers per particle, so 1.2Mb in total), and therefore the list version also gives relatively good speedups.

Speedups of the hybrid version of the N-Body simulation are given in 3.7(b). We can observe that the best speedup is obtained when using only one GPU. Adding CPU cores only slows down the execution. The reason for this is that the GPU is, for this particular computation, massively faster than the CPU cores,
therefore offloading any work to the CPU cores slows down the computation.

**Ant Colony Optimisation**  Ant Colony Optimisation (ACO) [4] is a heuristic for solving NP-complete optimisation problems. In this paper, 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, $p_i$ ($Ps$ in the code below), deadline, $d_i$ ($Ds$ in the code below), and weight, $w_i$ ($Ws$ 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 ($\tau$ in the code below). The pheromone trail is stronger along previously successful routes and is defined by a matrix, $\tau$, where $\tau[i,j]$ is the preference of assigning job $j$ to the $i$-th place in the schedule. After all ants compute their solution, 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:

```erlang
iterate_ants(_,_, _, _, _, _, 0) ->
    [];
iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, Tau, N) ->
    Scheds = lists:map(fun(X) ->
        find_solution(NrJobs, Ps, Ws, Ds, Tau)
    end,
    lists:seq(1, NrAnts)),
    {BestCost, BestSchedule} = pick_best:pick_best(Scheds),
    NewTau = update(NrJobs, BestCost, BestSchedule, Tau),
    iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, NewTau, N-1)).

ant_colony(InputFile, NrAnts, NrIters) ->
    {NrJobs, Ps, Ws, Ds, Tau} = init(InputFile),
    iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, Tau, NrIters).
```

The main function is `iterate_ants`, which is evaluated $N$ times. It first calculates `Scheds`, a list of schedules produced by individual ants. Then, it picks the running best schedule using `pick_best` function, updates the pheromone trail according to this schedule using `update` function, and then moves on to the next iteration. In the main function, `ant_colony`, an input is read into lists `Ps`, `Ws`, `Ds` and `Tau` using the `init` function, and then the `iterate_ants` function is called.

The overall structure of this application is similar to that of N-Body that was discussed in the previous section. We use a test input data with 100 jobs, using 64 000 ants. The initial parallelisation using Wrangler introduces a map skeleton in place of the original `lists:map` application in `iterate_ants`, using the `ParMapIntroSeq` refactoring. Ants are, therefore, grouped in chunks, with as many
chunks as there are CPU workers in the instance of the skeleton. Relevant parts of
the code, after refactoring, are given below:

chunk_ants(ListOfAnts, NrWorkers) ->
  ChunkSize = NrTasks div NrWorkers,
  Remainder = NrTasks rem NrWorkers,
  ChunkSizes = lists:duplicate(Remainder, {ChunkSize+1})
    ++ lists:duplicate(NrWorkers-Remainder, {ChunkSize}),
  ChunkSizes.

iterate_ants(_,_, _, _, _, _, 0) ->
  [];
iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, Tau, N) ->
  Map = {map, {func, fun(Chunk) ->
    process_chunk_ants(Chunk, NrJobs, Ps, Ws, Ds, Tau)
    end}},
  fun(ListOfAnts) -> chunk_ants(ListOfAnts, NrWorkers),
  fun (X) -> X},
  Scheds = skel:do([Map], [lists:seq(1,NrAnts)]),
  {BestCost, BestSchedule} = pick_best(Scheds),
  NewTau = update(NrJobs, BestCost, BestSchedule, Tau),
  iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, NewTau, N-1)].

ant_colony(NrWorkers, InputFile, NrAnts, NrIters) ->
  {NrJobs, Ps, Ws, Ds, Tau} = ant_init:init(InputFile),
  iterate_ants(NrAnts, NrJobs, Ps, Ds, Ws, Tau, NrIters).

The speedups for this version are given in Figure 3.8 as the “List version”. We
observe that this version gives very poor performance, compared to the original
sequential version. This is due to the fact that a large amount of data needs to
be copied between processes when this is represented in terms of lists. The en-
tire Ps, Ws, Ds and Tau lists need to be copied for each ant, which amounts to a
huge amount of data if the number of jobs is big. In the subsequent step, we have
therefore applied program shaping refactorings to produce both a version that uses
binaries and a version that uses ETS tables, using the ListToBinary and ListBina-
ryToETS refactorings (from Appendices A.1 and A.2, respectively). The speedups
of these two versions are also shown in Figure 3.8 as the “Binary version” and the
“ETS version”, respectively. We observe better speedups for these two versions,
especially for the ETS version, which gives a maximum speedup of 10 with 24
workers. The binary version gives worse speedups when run entirely on CPUs, but
provides a starting point for a hybrid version.

The next steps are to generate the code for the GPU workers and to introduce
hybrid skeletons (again, using the HybIntroMapSeq refactoring). Both of these
steps are similar to those for N-Body, since the structure of the code is similar.
Similar to particles in the N-Body example, a GPU kernel for Ant Colony can
compute solutions for multiple ants in parallel – with one logical GPU thread per ant. This allows us to get much more parallelism on a GPU, where we can process many ants in parallel, then on a CPU core where processing of ants is sequential. Using profiling information, we discovered that the GPU is capable of processing about 64 ants in parallel without sequentialisation, and that processing this many ants on the GPU is 25 times faster than processing them sequentially on a CPU core. Therefore, using mechanisms described in Section 3.4.3, we divide all ants into chunks, where the GPU chunk is a multiple of 64.

Speedups for the heterogeneous version of Ant Colony are given in Figure 3.8. As opposed to N-Body example, here we can see obvious benefits when combining CPU and GPU Workers. Using 1 GPU and no CPU workers gives a speedup of approximately 8.5, while using 1 GPU and 20 CPU workers gives a better speedup of 12.2. In this case, since total number of ants is 64000, and the GPU is “only” 25 times faster in processing a batch of 64 ants than a CPU, work division algorithm in Section 3.4.3 will allocate a lot of work to the CPU cores also, thus speeding up the computation compared to when only the GPU is used.

**Image Merge** Image Merge is an application from the computer graphics domain. It reads a stream of pairs of images from files, and merges images from each pair:

\[
\text{FinalImgs} = \left[ \text{convertMerge(readImage(Y))} \mid Y \leftarrow \text{imgList(X)} \right].
\]
There are multiple ways in which this code can be parallelised. Since the output of `readImage` is an input to `convertImage`, we can set up a pipeline between these two functions. Also, each of the two functions can be farmed, so we read and process multiple images at the same time. We will, however, consider the simplest parallelisation here: the one where we set up a `farm` of workers that execute `convertMerge . readImage`. For this, we use the `IntroduceTaskFarm` refactoring. The parallel code for this is:

```
skel:do([[farm, [[seq, fun (Y) ->
          convertMerge(readImage(Y)) end]],
         <nr_workers>]], imageList(X)).
```

This version, however, has problems with memory consumption. Since images are kept in memory as lists, for larger numbers of large images, the required memory was too high for our testbed machine. Therefore, we needed to apply program shaping refactorings to first represent images as binaries or ETS tables (the `List-ToBinary` and `ListBinaryToETS` refactorings). Figure 3.10 shows the speedups for the binary and ETS version of the code. We observe that both versions have very similar performance.

In the next step, we introduce a hybrid farm skeleton using the `HybFarmIntroSeq` refactoring from Section 3.5. The speedups for this version are also given in Figure 3.10. We can observe that, similar to the Ant Colony example, using both CPU and GPU workers gives better results than using only CPU workers. The best
speedup of 17.4 is obtained for 20 CPU workers and 1 GPU worker. Note that in this example we use the farm skeleton on a stream of images, so we assume that we do not have the whole input list of images available at the beginning. We, therefore, do not use any special division of work between the CPU and GPU workers. We simply assign images in round-robin fashion to the workers as they arrive.

**Football Simulation** Football Simulation is an industrial application provided by Erlang Solutions that predicts the outcomes of football games, and is used by betting companies to calculate the winning odds for matches. Each simulation accepts as arguments some information about the teams involved in a match (e.g. attack and defence strength), and uses a randomised routine to predict the end result of the match. The final prediction for each match is averaged over multiple simulations, so more time the simulation is repeated, more accurate the prediction will be.

The top-level structure of a code is

\[
\text{Results} = [\text{get\_score(sim\_match(Pair, NrSims))} || \text{Pair}<\text{Pairs}] 
\]

Pair is a pair of tuples that contains the necessary information about one match, i.e. information about one pair of teams. In the simplest case, we provide just two floating point numbers for each team, attack and defence strength. For each pair of teams, simulate_match is called NrSims times, and then the average score is
computed using the `get_score` function. We omit the details of the `sim_match` and `get_score` functions, as they are both inherently sequential.

We can observe that the potential parallelism lies at two levels of the computation of the `Results` list. At the outer level, we can parallelise the `get_score` function over the list of pairs of teams. At the inner level, for each match we do `NrSims` simulations, which are completely independent of one another and can, therefore, be executed in parallel. However, each individual simulation is very short, so parallelising the inner level results in too fine-grained parallelism. Therefore, we just parallelise the outer level. Similarly to Ant Colony and NBody examples, here we also use hybrid map skeleton. Furthermore, since `sim_match` operates over tuples of floating point numbers, rather than lists, we do not have binary and ETS versions.

Figure 3.11 shows the speedups we obtained on `titanic` for both CPU-only and hybrid version of Football Simulation. The figure shows speedups when different number of CPU workers in used in the map skeleton, with 0 GPU workers for the CPU-only, and 1 GPU worker for the hybrid version. We can observe that, for smaller number of workers, hybrid version significantly outperforms the CPU-only version. However, with the increase in the number of CPU workers, CPU-only version starts to perform similarly to the GPU version and even, for 22 and 24 CPU workers, outperforms it. The best speedup obtained is 17.4 with 1 GPU and 20 CPU workers.

We can see in this and other examples that the performance for hybrid versions tails when the number of CPU workers approaches the number of cores in the system. This is probably due to scheduling issues, as the GPU workers are
implemented using NIFs, which essentially lock up one scheduler for the whole duration of their execution, preventing any other Erlang processes being scheduled on them.
Chapter 4

ParaPhrase Refactoring Tool for Erlang

The ParaPhrase Refactoring Tool for Erlang, PaRTE [7] integrates a.) parallel pattern discovery, b.) candidate prioritisation and c.) semi-automated refactoring for shaping Erlang code and introducing algorithmic skeletons provided the skel library. Components of this tool have been already described in earlier deliverables. Here we present how the tool as a whole can be used (in Section 4.1), and describe two novel components of this transformation system:

- the cost-model based ranking of pattern candidates (in Section 4.2), and
- the interpreter of the static analysis and refactoring DSL (in Section 4.3).

PaRTE integrates capabilities of the RefactorErl [5, 22] and Wrangler [14] refactoring/program analysis tools into a new parallelisation framework that can be used to identify parallel patterns and determine the best implementations of those patterns. Both the pattern candidate detection/assessment and semantics-preserving transformation steps require thorough syntactic and semantic analysis. To do this, we exploit the RefactorErl and Wrangler refactoring tools, which implement specific compile-time analyses, and support syntax-based transformations. RefactorErl implements a wide range of static semantic analyses, including scope analysis of various language entities, side-effect analysis and callee approximation of dynamic function calls. Most of these (higher-level) semantic analyses build on results from dataflow and type analyses. Based on the information uncovered by the semantic analyses, RefactorErl can determine non-trivial properties of code fragments. In the integrated PaRTE framework, it is therefore used to perform pattern discovery and evaluation. Conversely, Wrangler has a mature user interface for performing simple code rewriting, which makes it a good choice for a tool to carry out parallelisation transformations. In the integrated PaRTE framework, a combination of Wrangler and RefactorErl is applied to define and execute shaping transformations, and to turn sequential computations into instances of parallel algorithmic skeletons.
4.1 The PaRTE framework

This section describes how to use the PaRTE tools, using Emacs to provide the user interface (in principle, other IDEs could be used, if preferred). The integration of the different tools and interfaces is seamless to the programmer, the installation as well as the execution of the tool chain is managed and supervised by PaRTE. Here we describe the approach taken by a single user on local Erlang source files (Figure 4.1). Since some of the PaRTE tools already allow multiple simultaneous users, it is reasonable to extend this to support collaborative development in the future.

**The Erlang developer starts Emacs, and enables the PaRTE mode.** This will start PaRTE. In more detail, the following will happen. Firstly, an Erlang node, called wrangler will be started: Emacs will communicate with this node. The wrangler node will start the Wrangler application, as well as another Erlang node, refactorerl, which will start the RefactorErl application, and introduce a server that is responsible for communication between Wrangler and RefactorErl.

**The developer opens an Erlang source file in Emacs.** Emacs notifies the wrangler node, which in turn notifies the refactorerl node. This loads the contents of the file into the RefactorErl database.

**The developer selects the Pattern discovery menu item.** Our tool analyses the freshly loaded code, collecting pattern candidates (see Deliverable D2.13 [19]).
Candidates satisfying the transformation side-conditions (Deliverable D2.12 [21]) will then be presented in the Pattern Candidate Browser (Deliverable D4.3 [18]) together with parallel execution time estimates and speedup predictions.

**A candidate is chosen by the developer in the Pattern Candidate Browser.** A pattern candidate specifies how a given code fragment can be transformed into a skeleton structure. For instance, a pattern candidate might specify that a farm of pipes should be introduced. The necessary transformation (which is often quite complex) is expressed as a sequence of smaller transformations, which include shaping transformations (see Deliverable D4.5 [20]) plus transformations to introduce seq, pipe, farm etc. skeletons.

**Transformations are performed under developer supervision.** Having selected a pattern candidate, PaRTE will carry out the required transformation sequence. A preview will be shown before major changes, offering the developer the possibility of cancelling the transformations and revert to the original state.

**The developer commits the changes.** Once the entire transformation sequence is completed, the developer may commit the changes to the source code. At this point, Emacs instructs the wrangler node to save the modified source code and instructs the refactorer node to reload and re-analyse these files.

Apart from the information provided by the Deliverables mentioned above, the user manual [11] of PaRTE describes the installation procedure of the tool as well as the user interface. Moreover, technical reports explain in more detail how pattern discovery is implemented [16], and how transformation rules are defined [6, 17].

### 4.2 Ranking with cost analysis

As mentioned, our pattern candidate browser displays code fragments which can be transformed into applications of paraphrase skeletons; and by an estimated speedup ratio (sequential execution time / parallel execution time) the browser ranks these pattern candidates, allowing the user to make good decisions on the possible parallelisation actions. In this section we introduce some details about the calculation of the speedup estimate, that is, the cost analysis phase of PaRTE.

#### 4.2.1 Compound pattern candidates

The pattern discovery phase finds source code segments that can be transformed into applications of parallel skeletons, and generates combinations of those pattern candidates that can be embedded into each other on the execution path. As a result,
we get transformation sequences: that is, collections of related elementary transformations that can be applied on a code fragment. Each transformation sequence consists of transformation descriptions, where a transformation description describes which code segment could be transformed into which kind of skeleton. In the cost analysis phase, we process every transformation sequence individually, and refine their transformation descriptions with the measured or calculated sequential and parallel execution time, and some additional information (e.g. the optimal number of workers in the case of a farm) if needed. In the case of a compound candidate, it is the outermost transformation description that will be displayed in the main table of the pattern candidate browser, since it contains information concerning the whole sequence.

4.2.2 Measuring and calculating sequential execution time

We have a benchmarking component that can measure the execution time of individual Erlang expressions. The expressions selected for benchmarking are put together with all their dependencies (functions, records, type specifications) so that they are encapsulated into an Erlang module. This module has only one exported function, parametrised by the free variables of the expressions in question. We invoke this function with a large number of randomly generated (well-typed) arguments. Function types are inferred by TypEr [15], and they are translated to QuickCheck [3] data generators controlling the distribution of test values. The average execution time that we obtain with these random values is taken to be the typical execution time of the expression.

We could measure the sequential execution time of a whole given candidate with the previously introduced benchmarking component, but rather we just benchmark the execution time of the work to be done on the stream data ($T_{stages}$ and $T_{work}$ in the case of pipelines and farms, respectively) and multiply it with the length of the stream. When the stream is given by a list, we try to calculate its length by following the dataflow in the source code, otherwise we use a default stream length. Both the sequential and parallel execution time (and hence, the speedup ratio) depend heavily on the length of the data stream, and if we benchmarked the whole candidate, the random generator would generate lists of different (and for the outer components – unknown) lengths. Instead, with our method, we can display in the candidate browser the data stream length used in our calculation, moreover, the user can adjust the default length.

Farm

In the case of a farm candidate, we have an Erlang expression that could be transformed to an application of a farm skeleton; and we also have a subexpression from which we can compose the worker function of the farm. Depending on the type of the candidate, we need to measure the execution time of the worker in different ways.
The simplest case is when we have a list comprehension candidate: its head expression has to be benchmarked.

*Example candidate:* \([f(X) \mid | X <- \text{List}]\)

*To be benchmarked:* \(f(X)\).

When we have an application of a lists:map function (or some other library function) as a farm candidate, its first actual parameter is the worker. This first parameter is a \((\text{fun}(A) \rightarrow B)\) type function, so the benchmarking component should create a function application from that, and generate A type input data for the execution.

*Example candidate:* \(\text{lists:map}(\text{fun}([X,Y]) \rightarrow X*Y \text{ end}, \text{List})\)

*To be benchmarked:* \(\text{fun}([X,Y]) \rightarrow X*Y \text{ end}(A)\).

The pattern discovery component can also find map-like recursive functions. In a recursive function, we cannot point out a subexpression that will be the worker of the farm, since it can only be synthesised by a rather complex shaping transformation. We do not want to apply any shaping transformations before benchmarking, therefore the benchmarking component rather inserts message sending instructions before (s message) and after (e message) the recursive call(s) of the function. These messages are received by a so-called timestamp evaluator component, which stores the messages with timestamps and evaluates them after the benchmarking component finished its work. This way we can calculate the average execution time of the body of the recursive function: 

\[
T_{\text{work}} = \frac{e_n - s_1}{n},
\]

where \(e_n\) is the timestamp belonging to the last e message, and \(s_1\) is the timestamp belonging to the first s message.

*Example candidate:*

\[
\begin{align*}
\text{r([])} & \rightarrow []; \\
\text{r([H | T])} & \rightarrow [f(H) \mid \text{r(T)}].
\end{align*}
\]

*To be benchmarked: r(X), where:*

\[
\begin{align*}
\text{r([])} & \rightarrow []; \\
\text{r([H | T])} & \rightarrow \\
& [ \begin{align*}
\text{ts_receiver} & ! s, \\
\text{Res} & = f(H), \\
\text{ts_receiver} & ! e, \\
\text{Res} & \\
\text{end}
\end{align*} \mid \text{r(T)}].
\end{align*}
\]

58
Pipeline

A pipeline candidate also contains the whole expression that can be transformed into the application of the pipelines skeleton, and we also have an expression list with the stages of the pipeline. Note that the stages need to be transformed into completely separate functions before we apply the skeleton introduction transformation, but, just like in the case of map-like recursive function, we do not want to apply any complex shaping transformations before benchmarking. In the source code there are many syntactically different occurrences of pipelines which would influence the evaluation of the result of the benchmarking, if we benchmarked the stages separately.

Examples

\[
\text{pipe(List)} \rightarrow [\text{st3(st2(st1(X))) || X <- List}].
\]

Here the execution time of the third stage includes the execution time of the second and the first stage. The execution time of the second stage also includes the execution time of the first.

\[
\text{pipe(List)} \rightarrow [\text{st1(X) || X <- List}].
\text{st1(X) -> ... st2(Y).}
\text{st2(X) -> ... st3(Y).}
\]

Here the first stage includes the second and the third, and the second includes the third.

\[
\text{pipe(List)} \rightarrow [\text{st1(X) || X <- List}].
\text{st1(X) -> ... st3(st2(Y)).}
\]

This is a mixed example: the first stage includes the second and the third, and the third includes the second.

We could benchmark the stages one by one, and then analyse the execution paths of expressions and calculate their separate execution time by subtracting the measured times from each other; but this way we would benchmark some subexpressions more than once unnecessarily. Instead, we chose to benchmark the whole pipeline candidate (e.g. the list comprehension), and insert message sending instructions before and after the expressions belonging to the stages – similarly to
the previously mentioned map-like functions. We tag the messages with the number of the stage. The timestamp evaluator component receives the messages and calculates the execution time of each separate stage.

The last example with message sending instruction

```
pipe(List) ->
  begin
    ts_receiver ! {s,1}, Res1 = st1(X),
    ts_receiver ! {e,1}, Res1
    end || X <- List.
  st1(X) -> ...
  begin
    ts_receiver ! {s,3},
    Res3 = st3(begin
      ts_receiver ! {s,2}, Res2 = st2(Y),
      ts_receiver ! {e,2}, Res2
      end),
    ts_receiver ! {e,3},
    Res3
  end.
```

Messages received by the timestamp evaluator component

\( t_1 : \{s, 1\}, t_2 : \{s, 3\}, t_3 : \{s, 2\}, t_4 : \{e, 2\}, t_5 : \{e, 3\}, t_6 : \{e, 1\} \)

Calculated execution time of the stages

\[ T_{stage1} = t_2 - t_1, \quad T_{stage2} = t_4 - t_3, \quad T_{stage3} = t_5 - t_4. \]

Compound

In the case of a compound pattern candidate, there is one or more embedded candidates in the worker of the outer farm, or in the stages of the outer pipeline. When calculating the execution time of the outer candidate, we keep the execution time of the embedded candidates already calculated with a fixed data stream length, and sum them with the execution time of the 'other' expressions inside the given worker or stage. We benchmark the outer job or stage while inserting message sending instructions before and after the embedded candidates, thus we will be able to assert the execution time of the 'other' expressions.
**Example**

\[ f(\text{ListOfLists}) \rightarrow [g(\text{List}) \mid \text{List} \leftarrow \text{ListOfLists}] \]. \$ farm0 \\
g(\text{List}) \rightarrow \\
... \$ exprs1 \\
\text{List2} = [h1(X) \mid X \leftarrow \text{List}], \$ farm1 \\
... \$ exprs2 \\
[h2(X) \mid X \leftarrow \text{List2}], \$ farm2

In our example there are three farm candidates, from which we can compose a compound candidate, where \$ farm1 \$ and \$ farm2 \$ are embedded by \$ farm0 \$. The execution time of the outer farm is:

\[ T_{farm0} = L \ast (T_{farm1} + T_{farm2} + T_{exprs1} + T_{exprs2}). \]

### 4.2.3 Calculating parallel execution time

For the calculation of the predicted parallel execution time we use a slightly modified version of the cost model published by the ParaPhrase consortium [9], in order to make our estimations more precise, namely, to cover implementation-level costs better. A calibration component checks both the process spawning and the heap copy speed of the actual machine, and we use these values in our cost model.

The cost model needs the following input data for estimating the parallel execution time of a candidate:

- Length of data stream: fixed when calculating the sequential execution time.
- Sequential execution time of the worker (farm) or the stages (pipeline).
- Number of cores: can be configured or queried from the system. In the case of compound candidates, the number of cores belonging to the outer candidate has to be divided among the embedded candidates.
- Number of workers (in the case of a farm): we calculate the parallel execution time for every possible number of workers within a well-founded interval, and we choose the worker number which results in the best predicted execution time.

### 4.3 DSL for refactorings and rule interpretation

The PaRTE framework not only discovers parallelisable code components, but it is also capable of semi-automatically transforming them into parallel equivalents. The tool implements several *refactoring* transformations, which reshape and turn the sequential code into a semantically equivalent, but parallel code.
Refactoring is essentially a source code transformation: a syntactically valid program is reshaped into another well-formed program. Syntactically correct code transformations are practically performed not on the source code level (i.e. the text), but on its model, the (abstract) syntax tree, so that transformations can be implemented in terms of tree transformations. (The model is turned back to the textual representation afterwards, in the process called “pretty-printing”.)

Refactorings typically consist of multiple rewriting steps: they may affect the source code at multiple different locations; that is, multiple individual subtrees of the model are changed during a transformation. Usually, rewriting reuses the parts of the “old” subtree in order to create the “new” subtree (e.g. a new expression, a new statement), such that parts of the new syntactic element are identical to parts of the old one.

There are rewritings where the new syntactic element does not depend on the old one (e.g. replacing every numeric constant by zero), but most rewritings in refactorings are not like that. Rather, in typical applications, the replacement contains parts of the original expression or statement.

For instance, when replacing additions by multiplications in a source code (formally, $X + Y$ by $X \times Y$, where $X$ and $Y$ are arbitrarily complex expressions), we clearly reuse the subtrees of the original operands.

A rewriting step is basically a three-phase process: match, construct and replace.

1. The syntactic element to be transformed is located and then matched against a specific pattern. Usually, some additional properties are checked as well.

2. The replacement is constructed, based on the replacement pattern typically referring to variables of the matching pattern.

3. The old subtree is replaced by the newly constructed replacement subtree.

Rewriting steps can be formalised in terms of rewriting rules, consisting of the match pattern, the conditions and the replacement pattern. In Deliverable D4.5 [20] we have already introduced and applied a formalisation technique to describe rewrite rules. Such a formal definition is not only usable as a specification/documentation: it can, and should, be executable as well. Therefore, we created a textual formalism (a domain specific language, DSL) for the rewrite rules, and we implemented an interpreter that can apply such rewrite rules on syntactically valid source code.

4.3.1 The Refactoring Language

As mentioned already, refactorings are performed on the syntax tree of the code: traversals, node matching and subtree creation is done via graph library calls, which is unnecessarily complex, as well as hard to write, read and maintain. In contrast,
the domain-specific language we introduce allows us to design and define refactoring transformations in terms of term rewriting rules, so that the matching pattern as well as the replacement pattern are written as \textit{Erlang syntactic schemes}, while conditions can be composed of various semantic \textit{predicates}. Note that this is mostly \textit{declarative}: we define what is replaced by what, rather than how.

\textbf{Rewrite rules} A program written in the refactoring language consists of a \textit{series of refactoring definitions}, where each refactoring is composed of arbitrarily many \textit{rewrite rules}. Rewrite rules in the DSL (see Figure 4.2) look very similar to the descriptions shown in Deliverable D4.5: the replacement pattern is put below the matching pattern (separated by a horizontal line like in an inference rule, or by a double arrow), while conditions may follow after the \textit{WHERE} keyword. (Rewrite rules without the replacement section are match-only rules: they may bind metavariables, but do not perform any tree modifications.)

Both the matching pattern and the replacement patterns are allowed to contain \textit{metavariables} (words starting with a capital letter are metavariables). Metavariables followed by a double dot (\_) are list metavariables which can match multiple subtrees (consecutive children of a node) or even no syntactic elements. When a bound metavariable is referred to in the replacement, the code that was matched with the matching pattern will be replicated and inserted during replacement construction.

The matching pattern is allowed to contain one or more syntactic schemes, so that consecutive children can be matched or constructed as replacement. Note that the syntactic scheme can only capture the \textit{context-free properties} of the tree (or trees) to be replaced; \textit{context-sensitive properties} of matched objects are checked in the conditions: by using the predicates defined in the semantic predicate library, we are able to express requirements on the static semantics of the syntactic elements.

\textbf{Rule modifiers} Rewrite rules can have modifiers which influence the execution strategy (e.g. whether separate target nodes are transformed in separate transactions, or in what ordering the target nodes are traversed) or the execution scope. We introduced a special modifier for restricting the syntactic scope of the rewrite rules: the \textit{IN} modifier allows to execute a rule only in a given subtree (identified
REFACTORING normalize_list_comp()
  [Head || GensFilters..]
  ------------------------
  List = [(Vars..) || GensFilters..],
  Fun = fun ((Vars..)) -> Head end,
  lists:map(Fun, List)
  WHERE
  Vars.. = intersect(boundVars(GensFilters..),vars(Head))

by a metavariable) rather than the whole syntax tree. Note that this expresses a
cost-sensitive condition on the syntactic elements to be transformed.

**Rule combinators** We stated that refactoring rules are composed of rewrite rules.
This is realised by using rewrite rule combinators: for instance, the THEN binary
combinator specifies that after successfully executing the first rule on a target node,
the second rule has to be run on its targets. The metavariables bound by the first
rule are visible in the second rule as well (and there is no name shadowing).

Although the individual rewrite rules are of declarative style, the refactoring
description itself is imperative: we define how rewrite rules are executed one after
the other. It is the responsibility of the refactoring writer to properly combine the
individual rewrite rules in order to get a semantics-preserving refactoring transfor-
mation in the end.

### 4.3.2 Examples

We have formalised (and therefore implemented) all the necessary shaping and
skeleton introduction refactoring transformations in the above methodology. In
this section, we show a simple and a less simple example showing how the rewrite
rules look like in practice.

The first example (Figure 4.3) demonstrates a refactoring step which can re-
shape any list comprehension into a trivial lists:map/2 application. The ele-
ments processed by the head of the comprehension are pre-generated into an aux-
iliary list, the operation executed on the list elements is extracted into a function,
and finally the comprehension is rewritten into a simple map call.

In this rule, one single expression is replaced by three other expressions. Worth
noting that GensFilters matches all the value generators and filter expressions
in the list comprehension, regardless how many and how complex they are. Vars
is a list metavariable bound not in the upper pattern but in the WHERE clause of
the rule, still it is allowed to refer to it in the replacement pattern (intersect, vars
and boundVars are semantic functions). Vars captures the variable names the
The second example demonstrates how rewrite rules are combined together to form a complex refactoring. The map-like function refactoring (Figure 4.4) takes a simple recursive function clause and turns it into a non-recursive clause showing the same behaviour, but by building upon the higher-order lists:map/2 function. This is beneficial because the map call in turn can be trivially converted into an instance of the farm skeleton.

Worth observing that this rule matches any simple recursive clauses having a list parameter, regardless the index of the list in the parameter list – this is achieved by using the two list metavariables (V1.. and V2..) around L, which can both match either zero, one or more elements in the parameter list. In this refactoring, first the compensation steps are executed in the clause body (because compensations depend on properties of the original function clause), and the extraction of the new kernel function as well as the actual rewriting to the map call is done in the last step of the transformation.

4.3.3 Rule Interpreter

The above detailed refactoring specifications are executed (more precisely, interpreted) by using the static analysis and graph transformation capabilities of the
RefactorErl tool, part of the PaRTE framework. The source code to be refactored is in fact loaded into the RefactorErl database and is manipulated in its model.

The matching and replacement patterns are Erlang syntactic schemes, they are barely valid Erlang code – thus one cannot use the standard parser or the RefactorErl parser to build their syntax tree. We employ a customised Erlang parser which not only accepts valid Erlang, but allows metavariables in place of almost any kind of syntactic construct, allowing us to parse the syntactic schemes.

When executing a rule on a target node, the rule interpreter performs matching by comparing the syntax tree returned by the customised parser with the tree belonging to the target node built in the RefactorErl framework. (Metavariables can match subtrees rather than just single nodes.) The result of the matching is the mapping of nodes to metavariables, which, due to having list metavariables, is non-deterministic: there can be multiple valid mappings. We require the rule conditions, however, to determine a single solution to the matching; if there are no solutions or there remain multiple solutions, the matching is failed and the refactoring is aborted. Once a metavariable is bound in a refactoring rule, it remains bound to the same value – refactoring rules use single assignment. Metavariables that are not bound prior to their usage are automatically initialised to fresh variable names.

The conditions are given in terms of first-order predicates, where most predicate symbols and function symbols are implemented by exploiting the information present in the RefactorErl semantic program graph. If the matching is successful and all the conditions are met, the system creates a syntactic subtree based on the replacement pattern. It is allowed to bind metavariables in the WHERE clause as well, they can be referred to in the replacement pattern. In the end, the old subtree is replaced by this new syntactic construct and the result can be printed back to Erlang text at the end of the transformation.
Chapter 5

A Refactoring Example: Agent Systems

We demonstrate how program shaping can be used by illustrating its application to the multi-agent system use case from D6.7. The system, developed by AGH, operates over a number of generations in finding a solution. Each generation may be modelled as an iteration of a loop, with each member of the system’s population performing its work within each iteration. Both the outer generational loop and the work performed within that loop are well suited for parallelisation. We include the code in question below, simplified for readability.

```erlang
loop(Islands, SP, Cf) ->
  receive
    the_end ->
      flatten(Islands)
    after 0 ->
      Groups =
        [group_by(
          [{behaviour_proxy(Agent, SP, Cf), Agent} || Agent <- I]
        || I <- Islands],
      Migrants =
        [seq_migrate(keyfind(migration, 1, Island), Nr)
          || {Island, Nr} <- zip(Groups, seq(1, length(Groups)))],
      NewGroups =
        [[meeting_proxy(Activity, mas_sequential, SP, Cf)
          || Activity <- I] || I <- Groups],
      WithMigrants = append(flatten(Migrants), NewGroups),
      NewIslands = [shuffle(flatten(I)) || I <- WithMigrants],
    loop(NewIslands, SP, Cf)
  end.
```

Whilst this code is a good candidate for parallelisation, it cannot be parallelised immediately, needing to first be shaped. We illustrate the method by which this
code may be shaped using existing refactorings. During the process, potential new shaping refactorings were also discovered, which we describe below.

5.1 New Shaping: Extract Comprehension Function

5.1.1 Motivation

Whilst the generic nature of skeletons allow their application to a number of problems, they must nevertheless be parameterised with problem-specific sequential code to allow them to be of any use. This problem-specific code is often in the form of a function, and that function usually already exists somewhere in the code base. Whilst task farms and maps can be extracted directly (D4.2) from list comprehensions, the programmer might want or require more elaborate workflows for greater efficiency, which often necessitates the extraction of the function from within a list comprehension. Separating the function from the comprehension allows greater flexibility in its use, and can allow it to be applied elsewhere whilst avoiding code duplication.

5.1.2 Description

This refactoring will extract the embedded on the left-hand side of a list comprehension, assigning it to a variable name provided for by the user. The list comprehension is transformed into a map, whose arguments consist of the newly assigned variable name, and the original input of the list comprehension.

5.1.3 Conditions

The refactoring is subject to a number of conditions, listed below.

1. The variable name provided by the programmer must be valid and not in scope.

2. The extracted function must be assigned to the variable in the same scope as the original list comprehension.

3. The list comprehension may only use one source list.

4. The outermost term of the left-hand side of the list comprehension must be a function call.

5. The refactoring may be used to extract functions from nested list comprehensions, but may only be applied from the innermost list comprehension to the outermost.
5.2 New Shaping: Compose Maps

5.2.1 Motivation
Functors are an important and often inescapable aspect of functional programming, with lists being perhaps the most prominent. Mapping over all elements of a list is a simple and concise means of affecting some change to that list. As such, is is not uncommon that lists will be transformed multiple times via a series of maps. When parallelising this series, composing the stages presented may provide greater efficiency over performing them in sequence.

5.2.2 Description
Reduces a series of maps to a single map operation that takes the original input and produces the original, eventual output by composing all functions, in order, extracted from the original series. The resulting composition is assigned to a variable, provided by the programmer, allowing reuse.

5.2.3 Conditions
The refactoring is subject to a number of conditions, listed below.

1. The variable name provided by the programmer must be valid and not in scope.
2. The resulting map operation must produce the same output as the original sequence.
3. All map operations in the sequence from the second to the last must take the immediate output of the preceding map.
4. The extracted function composition must be in the same scope as the original functions, allowing use of variables in scope not passed as a parameter.

5.3 Refactoring the Agent System
The rest of this chapter will use the above and other, pre-existing refactorings to transform the MAS to enable and introduce parallelism via the Skel library.

5.3.1 Stage 1
We start shaping `loop` by extracting functions from the list comprehensions assigned to `Groups`, producing the following code.
The functions assigned to TagFun and GroupFun are extracted from the inner and outer list comprehensions originally assigned to Groups using the *Extract Comprehension Function* refactoring described above. This shaping refactoring has also been applied to the function found in the list comprehension assigned to Migrants.

### 5.4 Stage 2

Upon examining the code we discover that the functions assigned to TagFun, GroupFun, and MigrantsFun can be composed. However, this composition first requires MigrantsFun to be changed to reflect a different input type. We first expand the body of MigrantsFun using the classical refactoring *Inline Method* in Wrangler, producing the following code.

```plaintext
loop(Islands, SP, Cf) ->
  receive
    the_end ->
    flatten(Islands)
  after 0 ->
    TagFun = fun(Agent) ->
      {behaviour_proxy(Agent, SP, Cf), Agent}
    end,
    Tagged = map(TagFun, Agents),
    GroupFun = fun(Island) ->
      group_by(Island)
    end,
    Groups = map(GroupFun, Tagged),
    MigrantsFun = fun(Island, Nr) ->
      seq_migrate(keyfind(migration, 1, Island), Nr)
    end,
    Migrants = map(EmigrantsFun, zip(Groups, seq(1, length(Groups))))
  NewGroups =
    [meeting_proxy(Acticity, mas_sequential, SP, Cf)
    || Activity <- I || I <- Groups],
  WithMigrants = append(flatten(Migrants), NewGroups),
  NewIslands = [shuffle(flatten(I)) || I <- WithMigrants],
  loop(NewIslands, SP, Cf)
end.
```
5.5 Stage 3

The result of the last transformation makes it clear that, currently, the output of Tagged cannot be streamed directly as input to MigrantsFun. However, the constituent parameters for MigrantsFun are the same as those of the output type of Tagged, and the operations applied thereto are similarly the same, allowing us to change the pattern match over the first clause of MigrantsFun.

This change allows us to compose TagFun, GroupFun, and MigrantsFun using the Compose Maps shaping. This transformation produces an anonymous function which we assign to TGM. The resulting code can be found below.

```haskell
loop(Islands, SP, Cf) ->
  receive
    the_end ->
      flatten(Islands)
  after 0 ->
    TagFun =
      fun(Agent) ->
        {behaviour_proxy(Agent, SP, Cf), Agent}
      end,
    Tagged = map(TagFun, Agents),
    GroupFun = fun group_by/1,
    Groups = map(GroupFun, Tagged),
    MigrantsFun =
      fun({{migration, Island}, From}) ->
        Destinations = map(fun getDestination/1, Island),
        GroupFun(Destinations);
        (OtherAgent) ->
          OtherAgent
      end,
    Migrants = map(EmigrantFun, zip(Groups, seq(1, length(Groups)))),
    NewGroups =
      [meeting_proxy(Activity, mas_sequential, SP, Cf)
        || Activity <- I || I <- Groups],
    WithMigrants = append(flatten(Migrants), NewGroups),
    NewIslands = [shuffle(flatten(I)) || I <- WithMigrants],
  end.
loop(NewIslands, SP, Cf)
```
5.6 Stage 4

We next apply *Extract Comprehension Function* to the list comprehensions assigned to *NewGroups* and *NewIslands*, producing the following code.

```
loop(Islands, SP, Cf) ->
  receive
    the_end ->
      flatten(Islands)
  after 0 ->
    TagFun =
      fun(Agent) ->
        {behaviour_proxy(Agent, SP, Cf), Agent}
      end,
    GroupFun = fun group_by/1,
    MigrantsFun =
      fun({{Home, migration}, Agent}) ->
        Destinations = map(fun getDestination/1, Agent),
        GroupFun(Destinations);
      end,
    OtherAgent
    TCM =
      fun(Agents) ->
        Tagged = map(TagFun, Agents),
        Migrants = map(MigrateFun, Tagged),
        Groups = GroupFun(Migrated),
      end,
    TCMs = map(TCM, Islands),
    NewGroups =
      [[meeting_proxy(Activity, mas_sequential, SP, Cf)
        || Activity <- I] || I <- TCMs],
    NewIslands = [shuffle(flatten(I)) || I <- NewGroups],
  loop(NewIslands, SP, Cf)
end.
```
loop(Islands, SP, Cf) ->
  receive
    the_end ->
      flatten(Islands)
  after 0 ->
    TagFun =
      fun(Agent) ->
        {behaviour_proxy(Agent, SP, Cf), Agent}
      end,
    GroupFun = fun group_by/1,
    MigrantsFun =
      fun({{Home, migration}, Agent}) ->
        Destinations = map(fun getDestination/1, Agent),
        GroupFun(Destinations);
        (OtherAgent) ->
          OtherAgent
      end,
    TGM =
      fun(Agents) ->
        Tagged = map(TagFun, Agents),
        Migrants = map(MigrateFun, Tagged),
        Groups = GroupFun(Migrated),
      end,
    TGMs = map(TGM, Islands),
    MeetingFun =
      fun(Activity) ->
        meeting_proxy(Activity, mas_sequential, SP, Cf)
      end,
    Meeting2Fun =
      fun(Island) ->
        map(MeetingFun, Island)
      end,
    NewGroups = map(Meeting2Fun, TGMs),
    NewIslandsFun =
      fun(Island) ->
        shuffle(flatten(Island))
      end,
    NewIslands = map(NewIslandsFun, NewGroups),
  end.
loop(NewIslands, SP, Cf)
5.7 Stage 5

We now have all elements necessary to build a Skel workflow. As the parameter Islands is to be our eventual input to Skel, we will encase the function TMG in a seq wrapper. We will also convert NewGroups and its associated functions into a seq wrapper; this conversion is aided by the IntroSeqMap refactoring described in D4.2. A task farm is employed to account for the nested map of NewGroups, producing the farm skeleton instance assigned to Map. The same refactoring is similarly applied to NewIslandsFun. These transformation produce the following code.

```
loop(Islands, SP, CF) ->
  receive
  the_end ->
    flatten(Islands)
  after 0 ->
    TagFun =
      fun(Agent) ->
        {behaviour_proxy(Agent, SP, CF), Agent}
      end,
    GroupFun = fun_group_by/1,
    MigrantsFun =
      fun({Home, migration, Agent}) ->
        Destinations = map(fun getDestination/1, Agent),
        GroupFun(Destinations);
        (OtherAgent) ->
          OtherAgent
      end,
    TMG =
      fun(Agents) ->
        Tagged = map(TagFun, Agents),
        Migrants = map(MigrateFun, Tagged),
        Groups = GroupFun(Migrants),
      end,
    TGMs = {seq, fun TGM/1},
    Work = {seq, fun({{Home, Behaviour}, Agents}) ->
      NewAgents =
        meetling_proxy({Behaviour, Agents}, new_skel, SP, CF),
        [{Home, A} || A <- NewAgents]
      end},
    Map = {farm, [Work], CF#config.skell_workers},
    Shuffle = {seq, fun(Agents) ->
      shuffle(flatten(Agents))
    end},
  end.
```

5.8 Stage 6

Next we invoke Skel using the workflow elements just constructed; a pipeline is built, Workflow, for this purpose. A list comprehension is added to ensure the
correct formatting of results. The outcome of this transformation can be found below.

```erlang
loop(Islands, SP, Cf) ->
    receive
        the_end ->
        flatten(Islands)
    after 0 ->
        TagFun =
            fun(Agent) ->
                behaviour_proxy(Agent, SP, Cf), Agent
            end,
        GroupFun = fun group_by/1,
        MigrateFun =
            fun({{Home, migration}, Agent}) ->
                Destinations = map(fun getDestination/1, Agent),
                GroupFun(Destinations);
                (OtherAgent) ->
                    OtherAgent
            end,
        TM =
            fun(Agents) ->
                Tagged = map(TagFun, Agents),
                Migrated = map(MigrateFun, Tagged),
                Groups = GroupFun(Migrated),
            end,
        TMs = {seq, fun TM/1},
        Work = {seq, fun({{Home, Behaviour}, Agents}) ->
            NewAgents =
                merging_proxy({Behaviour, Agents}, mas_skel, SP, Cf),
            [{Home, A} || A <- NewAgents]
        end},
        Map = {fann, [Work], Cf#config.skel_workers},
        Shuffle = {seq, fun(Agents) ->
            shuffle(flatten(Agents))
        end},
        NewIslands = [NewIsland
                        || (_, NewIsland) <- skel:do([Workflow], Islands)],
    loop(NewIslands, SP, Cf)
end.
```

### 5.9 Stage 7

The repeated invocation of Skel can introduce inefficiencies in the construction and destruction of Erlang processes. A feedback skeleton may instead be employed to replicate this outer loop, avoiding the repeated invocation of Skel. We introduce a constraint function, `Constraint`, to control the loop exit: this requires the variable `Time` to be added to the list of parameters. We replace the loop with a call to feedback within the Skel invocation.
5.10 Using Binaries

It is possible that the increased copying of lists introduced with parallelism will have a detrimental effect on the performance of the now-parallel MAS. Erlang’s binary data structure presents a means of avoiding the overheads associated with the copying of lists between processes. We might employ the List to Binary refactoring described in Section 3.5.2.1 to aid us in converting to the use of binaries.

To briefly illustrate this transformation process, we highlight a specific part of the example. The below function, `evaluation`, is used to update the solution during each generation.
evaluation(B, _SP) ->
   -lists:foldl(fun(X, Sum) ->
       Sum + 10 + X*X - 10*math:cos(2*math:pi()*X) end, 0.0, B).

Applying List to Binary, where B is the list/binary parameter, would necessitate a change to the fold operation. An equivalent fold that operates over binaries is introduced, with the type signature of binary elements provided by the user. The resulting code is given below.

evaluation(B, _SP) ->
   -bfoldl(fun(X, Sum) ->
       Sum + 10 + X*X - 10*math:cos(2*map:pi()*X) end, 0.0, B).

bfoldl(_, Init, <<>>) -> Init;
bfoldl(F, Init, <<H/float-signed, T/binary>>) ->
   bfoldl(F, F(H, Init), T).

5.11 Results

The plots in Figure 5.1 present the comparison of application performance using two different agent representations. In one we use normal erlang lists containing float numbers, in the other we used binary types instead. The reason behind this enhancement is that Erlang messages containing large binaries (>64B) are not copied between process heaps, but they reside in a separate memory segment therefore only references need to be copied. Due to the large amount of messages exchanged in the skel workflow, this change introduced a significant improvement in the application speed.

The plots in Figure 5.2 represent the performance boost that we have experienced after rearranging the skeletons in our workflow. Previously the whole workflow was encapsulated in the feedback skeleton which was responsible for stopping the algorithm after predefined time, however it also introduced a synchronisation barrier after each iteration. To be precise, we changed the order of skeletons from roughly {feedback, [{map, [OtherSkeletons]]}}, to {map, [{feedback, [OtherSkeletons]]}}. This change enabled each parallel map process to run asynchronously in its own loop and enabled our application to scale almost linearly.
Figure 5.1: Performance Results for Binary and List for Multi-Agent System
Figure 5.2: Performance Results for Further Tuned Multi-Agent System
Chapter 6

Conclusion

This deliverable has described the final phase of the Refactoring Tools implemented for the Pattern Transformation System, as described in T4.1. Specifically, in Chapter 3, we introduced the LAPEDO framework allowing programmers to easily write heterogeneous parallel programs in C++ and Erlang. LAPEDO contains new refactorings to introduce Hybrid skeletons (a skeleton containing a mixture of CPU and GPU components), automatic generation of the GPU components of hybrid patterns (including OpenCL code to create the necessary buffers on the GPU, transfer the data between the CPU and the GPU memory and schedule the execution of GPU kernels on GPU devices), Program Shaping refactorings (such as List to Binary for Erlang) and also a static allocator to divide tasks into CPU and GPU resource bound tasks at runtime. Specifically, in Section 3.7 we gave a number of use-cases (including the Molecular Dynamic from HLRS and Football Simulation use-case from our Industrial Partner, Erlang Solutions), and showed how we could parallelise each use-case for a heterogeneous system, using LAPEDO. In Chapter 4, we introduced PaRTE, the Parallel Refactoring Tool for Erlang, which includes an automatic profile and discovery system, plus refactorings from D4.1. In Chapter 2 we introduced new formal reasoning for our refactoring system, based on a set of operational and denotational semantics, we gave soundness proofs for some Erlang refactorings. Finally, in Chapter 5, we refactored the AGH use-case (from WP6), applying our program shaping refactorings and skeleton introductions.

The use of our technology on industrial use cases shows that we are able to obtain, for realistic applications, very good speedups while significantly reducing the effort required for parallelisation. The LAPEDO framework requires only the identification of the appropriate high-level parallel structure of an application, based on the framework guiding, semi-automatically, the programmer in parallelising the application. Pattern discovery tools offer further assistance in this process of application parallelisation, by automatically discovering the candidate parallel structures that can be applied to the application, offering at the same time the estimation of the performance of each of them on a given hardware. Therefore, the programmer is only required to select one of the offered parallel structures, and our
Table 6.1: Approximate implementation time, manual vs. refactoring

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>3 days</td>
<td>3 hours</td>
<td>58</td>
</tr>
<tr>
<td>Ant Colony</td>
<td>1 day</td>
<td>1 hour</td>
<td>32</td>
</tr>
<tr>
<td>BasicN2</td>
<td>5 days</td>
<td>5 hours</td>
<td>40</td>
</tr>
<tr>
<td>Graphical Lasso</td>
<td>15 hours</td>
<td>2 hours</td>
<td>53</td>
</tr>
</tbody>
</table>

framework does all of the “dirty work” required to apply that structure to a given application. This almost completely eliminates the need for dual-expertise, therefore not requiring application programmers to also be system programmers. Our work also represents a crucial step towards the ideal scenario of a fully-automatic parallelisation of user applications. Furthermore, formal proofs of soundness of the transformations carried out by the LAPEDO framework give further confidence in the correctness of the transformed applications.

The use of Pattern Discovery and Program Shaping shows that refactoring systems to simply introduce parallel skeletal code are not enough. Most often, applications are simply not in the desired shape to introduce the parallel skeleton. This may be because of inherent design choices such as dependencies, global state or choices in particular data types (Erlang’s list versus binary structure, for instance). However, we have demonstrated that our refactoring approach can go far beyond merely introducing parallelism structures into applications. We also extended our refactoring tool to be able to change the shape of a program so that the skeletons can then be introduced by further refactoring almost trivially. This is a crucial step to bridge the gap between legacy applications and programmers writing in the old think sequential style, to the new think parallel approach.

Refactoring gives an enormous saving in effort. Table 6.1 shows approximate porting metrics for each use case reported in D6.4, with the time taken to implement both manually (by a FastFlow expert) by our industrial partners, and with the aid of the refactoring tool. Clearly, the refactoring tool gives an huge saving in effort over manual implementation of the FastFlow code.

In the recently started RePhrase EU H2020 project, we will extend the work done in ParaPhrase for C++ by developing a pattern transformation system for the most popular parallel programming models and libraries, such as OpenMP, pthreads and Intel TBB. While ParaPhrase was focused on implementation and deployment phases of the typical application lifecycle, RePhrase will also produce tools to help with or automate other software engineering phases such as requirements analysis, design, debugging, testing and maintenance. In the future, we also plan to extend the ParaPhrase tools and techniques to more-heterogeneous systems, including the ones that comprise FPGA accelerators and specialist processors, such as DSPs. We will also adapt the tools to integrated and embedded systems, further extending the transformation system to take into account metrics
other than performance, such as energy efficiency, cost-effectiveness and/or costs of data-movements. Finally, we plan to extend the transformation system to add more flexibility and adaptivity to applications, enabling them to adapt statically and dynamically to the changing execution environment or runtime requirements.
Appendix A

Full list of Refactoring Rules

A.1 List to Binary

\[
\text{ListToBinary}(\rho, f, \text{Arg}, T, \text{types}) = \\
\mathcal{D}[f] \Rightarrow [f(\overline{x}, \text{Arg}', \overline{y}) \rightarrow \text{body}'] \\
\{ f \in \rho, \overline{x} \in \rho, \overline{y} \in \rho, \text{Arg} \in \rho, T \notin \rho \}
\]

where

\[
\text{Arg}' = \text{ListToBinaryExpr}(\rho, \text{Arg}, T, \text{types}) \\
\text{body}' = \text{ListToBinaryExpr}(\rho, \text{body}, T, \text{types}) \\
f = [f(\overline{x}, \text{Arg}, \overline{y}) \rightarrow \text{body}]
\]
\[
\text{ListToBinaryExpr}(\rho, \text{Arg}, T, \text{\textsc{types}}) = \]
\[\mathcal{E}\left[\text{\textsc{Arg}}\right] \Rightarrow [T = g'(\overline{y}), \langle< \overline{\mathcal{X}}, T/\text{binary} >>] \]
\{\text{Arg} \in \rho, g \in \rho, \overline{x} \in \rho, \overline{y} \in \rho, T \not\in \rho\}
where
\[g' = \text{ListToBinaryExpr}(\rho, \text{Arg}, T, \text{\textsc{types}}), \text{Arg} = \left[\overline{x} | g(\overline{y})\right]K\]
\[
\oplus\mathcal{E}[e] \Rightarrow [n : h(\overline{x}, \text{Arg}, \overline{z})] \{\overline{\mathcal{X}} \in \rho, \overline{\mathcal{Z}} \in \rho, g \in \rho\}
where
\[n : g = \text{ListToBinaryExpr}(\rho, \text{Arg}, T, \text{\textsc{types}}), e = [m : g(\overline{x}, \text{Arg}, \overline{z})]\]
\[
\oplus\mathcal{E}[e] \Rightarrow [g'(\overline{x}, \text{Arg}', \overline{z})] \{g \in \rho, \overline{x} \in \rho, \overline{z} \in \rho\}
where
\[g' = \text{ListToBinaryExpr}(\rho, \text{Arg}, T, \text{\textsc{types}}),
\text{Arg}' = \text{ListToBinaryExpr}(\rho, \text{Arg}, T, \text{\textsc{types}}),
e = [g(\overline{x}, \text{Arg}, \overline{z})]\]
\[
\oplus\mathcal{E}[e] \Rightarrow [\text{\textsc{Arg}'}] \{e \in \rho\}
where
\[\text{Arg}' = \text{ListToBinaryExpr}(\rho, \text{Arg}, \text{\textsc{types}})\]
\text{ListToBinaryExpr}(\rho, \text{Arg}, \text{types}) = \\
E[e] \Rightarrow \{ \text{lists : } y(a, \text{binary_to_list(Arg)}) \} \{ \text{Arg} \in \rho, a \in \rho, y \in \rho \} \\
\text{where} \\
y = [\text{map}] \vee y = [\text{foldl}], \ e = [\text{lists : } y(a, \text{Arg})] \\
\oplus E[\text{Arg}] \Rightarrow \llbracket <\text{types}, \text{binary} >\rrbracket \{ \text{Arg}, T \in \rho, \vec{\text{x}} \in \rho \} \\
\text{where} \\
\text{Arg} = \llbracket [\vec{\text{x}} \mid T] \rrbracket \\
\oplus E[\text{Arg}] \Rightarrow \llbracket x \rrbracket \{ \text{Arg} \in \rho, x \in \rho \} \\
\text{where} \\
\text{Arg} = \llbracket \text{binary_to_list(x)} \rrbracket \quad (A.3) \\
\oplus E[\text{Arg}] \Rightarrow \llbracket <\vec{\text{x}} >\rrbracket \{ \text{Arg} \in \rho, \vec{\text{x}} \in \rho \} \\
\text{where} \\
\text{Arg} = \llbracket [\vec{\text{x}}] \rrbracket \\
\oplus E[\text{Arg}] \Rightarrow \llbracket << >\rrbracket \{ \text{Arg} \in \rho \} \\
\text{where} \\
\text{Arg} = \llbracket [] \rrbracket \\
\oplus E[\text{Arg}] \Rightarrow \llbracket \text{Arg} \rrbracket \{ \text{Arg} \in \rho \}
A.2 List or Binary to ETS Table

\[
\text{ListBinaryToETS} \(\rho, M, f, r, d, i, \text{TName}, \text{IName}, \text{Term}\) = \\
\text{IntroCreateFun} \(\rho, M, \text{TName}\) \circ \text{WrapWithCAndR} \(\rho, f, r\) \circ \\
\text{IntroIDWrapperFun} \(\rho, i, \text{Term}\) \circ \text{TabIntro} \(\rho, d, \text{TName}, \text{IName}\)
\]
\[
\{M \in \rho, f \in \rho, d \in \rho, r \in \rho, i \in \rho, \text{Term} \in \rho, \text{IName} \notin \rho, \text{TName} \notin \text{tables}(\rho)\}
\]

\[
\text{TabIntro} \(\rho, d, \text{TName}, \text{IName}\) = \\
\text{InsertRetToETS} \(\rho, d, \text{TName}, \text{IName}\) \circ \text{WrapWorkerFun} \(\rho, g, \text{TName}, \text{IName}\)
\]
\[
\{d \in \rho, \text{TName} \in \text{tables}(\rho), \text{IName} \in \rho\}
\]
where
\[
g = d \mapsto g
\]

\[
\text{IntroCreateFun} \(\rho, M, \text{TName}\) = \\
\mathcal{M}[[M]] \mapsto [[\text{macr} \circ \text{create}() \rightarrow \text{TName} = \text{ets} : \text{new}(\text{TName}, \{\text{set, public, named_table,} \}
\{\text{write_concurrency}, \text{true}, \{\text{read_concurrency}, \text{true}\}})]] \mapsto \text{dec}]]
\]
\[
\{M \in \rho, \text{macr} \in \rho, \text{dec} \in \rho, \text{create/0} \notin \text{decs}(\rho), \text{TName} \notin \text{tables}(\rho)\}
\]
where
\[
M = [[\text{macr} \circ \text{create}() \rightarrow \text{dec}]]
\]

\[
\text{WrapWithCAndR} \(\rho, f, r\) = \\
\mathcal{D}[[x]] \mapsto [[f(y) \mapsto \text{create}(), \text{body} \rightarrow r] \{f \in \rho, r \in \rho, y \in \rho, \text{body} \in \rho\}
\]
where
\[
x = [f(y) \mapsto \text{body}]
\]

\[
\text{IntroIDWrapperFun} \(\rho, i, \text{Term}\) = \\
\mathcal{E}[[x]] \mapsto [[i(\text{Term})] \{i \in \rho, \text{Term} \in \rho\}
\]
where
\[
x = \text{Term}
\]

\[
\text{InsertRetToETS} \(\rho, d, \text{TName}, \text{IName}\) = \\
\mathcal{D}[[d]] \mapsto [[d(\{\text{IName}, x\})] \mapsto \text{body}, \text{ets} : \text{insert_new}(\text{TName}, \{\text{IName, body}\})], \text{IName}]
\]
\[
\{d \in \rho, \text{TName} \in \text{tables}(\rho), \text{IName} \in \rho, x \in \rho, \text{body} \in \rho, \text{body} \in \rho\}
\]
where
\[
d = [[d(x) \rightarrow \text{body}, \text{body} \rightarrow ]]
\]

\[
\text{WrapWorkerFun} \(\rho, g, \text{TName}, \text{IName}\) = \\
\mathcal{D}[[g]] \mapsto [[g(\text{IName}) \mapsto [[\text{IName, x}] = \text{ets} : \text{lookup}(\text{TName}, \text{IName}), \text{body}, \text{ets} : \text{insert}(\text{TName}, \{\text{IName, body}\}), \text{IName}]
\]
\[
\{g \in \rho, \text{TName} \in \text{tables}(\rho), \text{IName} \in \rho, x \in \rho, \text{body} \in \rho, \text{body} \in \rho\}
\]
where
\[
g = [[g(x) \rightarrow \text{body}, \text{body} \rightarrow ]]
\]
A.3 Introduce Hybrid Map Rules

\[ \text{HybMapIntroSeq}(\rho, e, g, n_{CPU}, n_{GPU}, s, com) = \]
\[ \mathcal{E}[e] \Rightarrow \left[ \{ \text{map}, \{ \text{seq, fun}(x) \rightarrow \right. \]
\[ \text{het_map : het_dispatcher(lists : map(fun(y) \rightarrow c', y)), g, x) \text{ end}] \} \]
\[ \{ \text{seq} \in \rho, \text{map} \in \rho, g \in \rho, x \text{ fresh}, y \text{ fresh}, \text{com} \in \rho, s \in \rho, c \in \rho, \]
\[ \text{het_map} \in \text{imports}(\rho), \text{het_dispatcher} \in \rho, \text{het_split} \in \rho, \text{args} = \text{free}(c, \rho) \} \]
\[ \text{where} \]
\[ c' = \left[ \text{fun}((\text{args}) \rightarrow c \right], e = \left[ \{ \text{seq, c} \} \right] \]

\[ \mathcal{E}[e] \Rightarrow \left[ \{ \text{skel : do}([\{\text{map}, \{\text{seq, fun}() \rightarrow} \right. \]
\[ \text{het_map : het_dispatcher(lists : map(fun(y) \rightarrow c, y)), g, x) \text{ end}] \} \}, \]
\[ \{ \text{seq} \in \rho, \text{map} \in \rho, g \in \rho, x \text{ fresh}, y \text{ fresh}, c \in \rho, \text{split} \in \rho, \text{com} \in \rho, \]
\[ \text{het_map} \in \text{imports}(\rho), \text{het_dispatcher} \in \rho, \text{het_split} \in \rho, \]
\[ \text{skel} \in \text{imports}(\rho), \text{do} \in \rho, \text{inputs} \in \rho \} \]
\[ \text{where} \]
\[ e = \left[ \{ \text{lists : map(c, Inputs)} \} \right] \vee e = \left[ \{ \text{lists : map(c, Inputs)} \} \right] \]

\[ \text{HybMapIntro}(\rho, e, g, n_{CPU}, n_{GPU}) = \]
\[ \mathcal{E}[e] \Rightarrow \left[ \{ \text{map}, \{ \text{seq, fun}(x) \rightarrow \right. \]
\[ \text{het_map : het_dispatcher(c', g, x) \text{ end}] \}, \]
\[ \text{fun}(x) \rightarrow \text{het_map : het_split(s, x, n_{CPU}, n_{GPU}) \text{ end}, com} \} \]
\[ \{ \text{seq} \in \rho, \text{map} \in \rho, g \in \rho, x \text{ fresh}, c \in \rho, s \in \rho, \text{com} \in \rho, \]
\[ \text{het_map} \in \text{imports}(\rho), \text{het_dispatcher} \in \rho, \text{het_split} \in \rho, \]
\[ \text{free}(c, \rho) \} \]
\[ \text{where} \]
\[ c' = \left[ \text{fun}((\text{args}) \rightarrow c \right], e = \left[ \{ \text{map, c, s, com} \} \right] \]

A.4 Parallel Map Introduction

\[ \text{ParMapIntroSeq}(\rho, e, g, p, c) = \]
\[ \mathcal{E}[e] \Rightarrow \left[ \{ \text{map, \{seq ?MODULE : g/1]} \}, \text{fun ?MODULE : p/1,} \right. \]
\[ \text{fun ?MODULE : c/1} \} \]
\[ \{ \text{seq} \in \rho, \text{parmap} \in \rho, p \in \rho, c \in \rho, g \in \rho \} \]
\[ \text{where} \]
\[ e = \{ \text{seq, F} \} \]

\[ \text{ParMapIntroComp}(\rho, e, g, p, c) = \]
\[ \mathcal{E}[e] \Rightarrow \left[ \{ \text{skel : run}([\{ \text{map, \{seq ?MODULE : g/1]} , \right. \]
\[ \text{fun ?MODULE : p/1, fun ?MODULE : c/1} ]], \text{Inputs}) \} \]
\[ \{ \text{skel} \in \text{imports}(\rho), \text{parmap} \in \rho, \text{run} \in \rho, \text{seq} \in \rho, \text{inputs} \in \rho, \]
\[ p \in \rho, c \in \rho, g \in \rho \} \]
\[ \text{where} \]
\[ e = \left[ \{ \text{f(Input)} \right| | \text{Input} \leftarrow \text{Inputs} \} \right] \]

87
A.5 Introduce Task Farm Refactoring

$$FarmIntroSeq(\rho, e, Nw) =$$

$$\{\text{farm} \in \rho, \text{seq} \in \rho\}$$

where

$$e = \{\text{seq}, E\}$$

$$FarmIntroMap(\rho, e, Nw) =$$

$$\{\text{seq} \in \rho, \text{farm} \in \rho\}$$

where

$$e = \text{lists : map} (\text{fun } \text{MODULE : f/1, List})$$

$$FarmIntroComp(\rho, e, Nw) =$$

$$\{\text{skel} \in \text{imports}(\rho), \text{run} \in \rho, \text{seq} \in \rho, \text{farm} \in \rho\}$$

where

$$e = [f(\text{Input}) \mid\mid \text{Input} \leftarrow \text{Inputs}]$$

$$FarmIntroComp2(\rho, e, Nw) =$$

$$\{\text{skel} \in \text{imports}(\rho), \text{run} \in \rho, \text{seq} \in \rho, \text{farm} \in \rho, \text{Input fresh}\}$$

where

$$e = [f_1(f_2(\ldots f_n(\text{Input})\ldots)) \mid\mid \text{Input} \leftarrow \text{Inputs}]$$

(A.7)
Bibliography


