Project no. 288570

PARAPhrase

Strategic Research Partnership (STREP)
PARALLEL PATTERNS FOR ADAPTIVE HETEROGENEOUS MULTICORE SYSTEMS

Pattern Amenability
D2.10

Due date of deliverable: 31st December 2013

Start date of project: October 1st, 2011

Type: Deliverable
WP number: WP2
Task number: T2.1

Responsible institution: ELTE
Editor and editor’s address: D. Horpácsi, J. Kőszegi, ELTE; V. Fördős, ES

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

This deliverable reports on the exemplars and findings we collected and made related to the possibilities of automatic discovery of Paraphrase patterns in Erlang programs. In our examination, we have considered both low-level and application-specific patterns as defined in deliverables D2.1 [6] and D2.3 [5]. Nevertheless, in case of some low-level patterns, such as Farm and Map, in order to comply with the present implementation, the Erlang parallel skeleton library, we have worked with the Erlang-specific variants of the patterns.

We composed exemplars for each Paraphrase pattern, bearing in mind that these samples will be used to assess, design and implement automatic discovery of parallel patterns. Therefore, in the case of core patterns, in addition to giving executable code snippets, we also defined the syntactic scheme of code amenable to automatic identification and transformation. Such generic definitions can significantly ease the forthcoming phases of T2.4, including the complete identification of patterns and the determination of transformation side-conditions. On the other hand, in the case of high-level and application-specific patterns, we only gave simple Erlang exemplars of the patterns, in order to show how such patterns would be realised in functional languages. We expect that higher-level patterns will be effectively identified from instances of low-level patterns rather than from sequential code.
Positioning of Deliverable D2.10

The positioning of this deliverable with respect to other deliverables in the work package can be seen in Figure 1. When preparing D2.10, we considered the patterns defined in D2.1 and D2.3, while the results of this work will be mainly utilised in D2.12 (and have been partly taken into account in D2.11 [8]).

Figure 1: Positioning of deliverable D2.10 w.r.t. other WP2 deliverables
# Contents

Executive Summary ................................................. 1

1 Introduction ......................................................... 4

2 ParaPhrase Pattern Exemplars ...................................... 5
   2.1 Core Patterns .................................................. 5
      2.1.1 Farm ...................................................... 5
      2.1.2 Pipeline .................................................. 11
      2.1.3 Map ....................................................... 15
      2.1.4 Reduce ................................................... 15
   2.2 High-level and Application-specific Patterns .................. 17
      2.2.1 Divide and Conquer ...................................... 18
      2.2.2 Pool Evolution .......................................... 19
      2.2.3 Orbit ..................................................... 20
      2.2.4 Possibly Iterated Numerical Library Calls .............. 20
      2.2.5 (Colored) Stencil ........................................ 21
   2.3 Patterns in Parallel Programs ................................ 22
      2.3.1 RPC Calls ................................................ 23
      2.3.2 Hand-made parallelism ................................... 23
      2.3.3 Hand-made skeletons ..................................... 24
      2.3.4 Distributed applications ................................. 25
Chapter 1

Introduction

In this document, we enumerate and describe a series of Paraphrase pattern exemplars written in Erlang, aiming at pointing out the (syntactic and semantic) characteristics of code possibly rewritable to Paraphrase parallel skeletons. In order to reveal the possibilities of automatic discovery of such patterns, in some cases we also defined syntax schemes that show common properties of transformable code.

Since the goal of T2.4 is to identify sequential programs that are transformable to parallel patterns, most of our examples present sequential Erlang code. In Section 2.1 we discuss the core patterns, while in Section 2.2 we elaborate the high-level and application-specific patterns. In addition to sequential exemplars, we also addressed the question of paraphrasing programs that already contain some sort and level of (ad-hoc) parallelism (see Section 2.3).

Throughout this document, boxes with white and yellow background contain sequential and ad-hoc parallelized code, while code in green boxes show our suggestions to rewriting to skeletons. The yellow-backed code and the green-backed code are sequential and parallel variants of the white-backed sequential code, and show the same semantics and observable behaviour. This notation of exemplars suggests that, in every exemplar, the white and the yellow ones can be transformed to the green one without making any changes in the program behaviour.
Chapter 2

ParaPhrase Pattern Exemplars

In this chapter, we show several Erlang exemplars for the Paraphrase patterns. For each category of sequential code scheme, we show a transformed, parallelised version that builds upon the Skel library [3, 4], the Erlang implementation of the core patterns.

2.1 Core Patterns

In this section, a wide range of exemplars are shown for the core patterns defined in D2.1. Where feasible, we also constructed syntax schemes for the examples, depicting some common structure.

2.1.1 Farm

One of the most obvious data parallel patterns: some computation executed on a long series of data items.

Syntactically, map-like patterns (list comprehensions, maps, recursive functions) that apparently perform the same computation on every element of a data list.

Note that the farm skeleton, in the current implementation, does not preserve the order. Therefore, the refactoring should ask the user whether an “Ord” wrapper is to be added or not to the workflow to keep the original order of the elements.
List comprehensions

**Syntax scheme**

```
[ ... P ... || ... , P <- ... , ... ]
```

Any kind of expression parameterized by elements of pattern \( P \) on the left, \( P \) in the head of a generator on the right (which should be independent of preceding generators). For the sake of simplicity, in the following examples, the principle generator is a simple variable named \( \text{List} \).

Note: we do not deal with the cases in which the head does not refer to \( P \) at all. That would be task parallel rather than data parallel.

```
[do_something() || _ <- List]
```

Let us define the following auxiliary functions.

```
id(X) -> X.
skelfarm(Fun, List) -> skel:do([\{farm, [\{seq, Fun\}]\}], List).
```

First, let us consider cases where there are no filters in the comprehensions. There can be multiple generators, though. In the latter case, we parallelize based on a principal generator (at least in one single transformation).

**FARM1: List comprehension with one generator**

```
[foo(X) || X <- List]
```

```
lists:map(fun(X) -> foo(X) end, List)
skelfarm(fun foo/1, List)
skelfarm(fun(X) -> foo(X) end, List)
```

**FARM2: List comprehension with one generator (more complex head)**

```
[case X of ... end || X <- List]
```

```
skelfarm(fun(X) -> case X of ... end end, List)
```
FARM3: Multiple generators

The head of the comprehension can freely refer to any of the items produced by additional generators. During the transformation, they are either embedded into the parallelized part, or the data records are pre-generated as tuples.

\[
\text{List3} = \{X, Y\} \mid X \leftarrow \text{List}, Y \leftarrow \text{List2}, \\
\text{skelfarm} (\text{fun}(X) \rightarrow [\text{foo}(X, Y) \mid Y \leftarrow \text{List2}] \text{ end, List})
\]

FARM4: Additional generators

Multiple additional generators can be handled in the following way.

\[
\text{skelfarm} (\text{fun}(X) \rightarrow [\text{foo}(X, Y, Z) \mid Z \leftarrow \text{List1}, Y \leftarrow \text{List2}] \text{ end, List})
\]

Filters of any form are allowed. Those referring to the items produced by the principal generator can be included in the parallelized part.

FARM5: Filters

\[
\text{List2} = \{X\} \mid X \leftarrow \text{List, filter(X)}, \\
\text{lists:flatten} (\text{skelfarm} (\text{fun}(X) \rightarrow [\text{foo}(X) \mid \text{filter(X)}] \text{ end, List}))
\]

If we pre-generate the filtered list and then parallelize based on the list comprehension, the filtering will not be done in parallel. In contrast, if we build upon the proposed solution in FARM5, we get a fully parallel version, because the filtering is done inside the farm skeleton. In the latter case, however, all the data pieces are sent over the network, even if they do not comply with the filter, which can result in a significant communication overhead.
Higher-order functions on lists

Here we take three well-known higher-order list functions into account: map, foreach and filter. The execution of these can easily be turned into data-parallel provided that some conditions are met.

### Syntax scheme

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>lists: map( ... , ... )</td>
</tr>
<tr>
<td>foreach</td>
<td>lists: foreach( ... , ... )</td>
</tr>
<tr>
<td>filter</td>
<td>lists: filter( ... , ... )</td>
</tr>
</tbody>
</table>

Parameters can be of any syntactic form, the transformation keeps them as is.

#### FARM6: Applications of `lists:map/2`

*The function can be given either as an implicit fun or as an explicit anonymous function.*

```erlang
lists:map(fun foo/1, List),
lists:map(fun(X) -> foo(X) end, List),
skelfarm(fun foo/1, List),
skelfarm(fun(X) -> foo(X) end, List),
```

#### FARM7: Nested maps

*See FARM3*

```erlang
lists:map(
  fun(X) ->
    lists:map(fun(Y) -> foo(X, Y) end, List2),
  end, List)

skelfarm(fun(X) ->
  lists:map(fun(Y) -> foo(X, Y) end, List2)
end, List)

skelfarm(fun(X) -> [foo(X, Y) || Y <- List2] end, List)
```

The farm skeleton does not respect the order of elements. If the user wants to keep the order of side-effects, an “Ord” wrapper needs to be introduced.
(Tail-recursive) Implementations of map

In this case, the observation is to be made not on expressions, but on Erlang forms.

```erlang
map1(_, []) -> [];  
map1(F, [H|T]) -> [F(H) | map1(F, T)].
map1(F, L) -> skelfarm(F, L).
```

Alternatively, once we found out that map1 behaves like lists:map, we can transform the calls to that function.

```erlang
map1(fun foo/1, List)  
skelfarm(fun foo/1, List).
```

Map-like functions

It is apparent that the patterns are semantic rather than syntactic phenomenons. Consequently, if calls to lists:map can be turned into farms, any code behaving like a map (i.e. having similar data and control flow) can be turned into a farm.
### Syntax scheme

```plaintext
mapfoo([], ...) -> [];
mapfoo([H | T], ...) when ... -> ...
  [... | mapfoo(T, ...)];
...
mapfoo([H | T], ...) when ... -> ...
  [... | mapfoo(T, ...)].
```

### Syntax scheme (2)

```plaintext
mapfoo(List, ...) ->
  case ...List... of
  ...[...]... -> [];
  ...[H|T]... when ... -> ..., [... | mapfoo(T, ...)];
  ...
  ...
  ...
  ...[H|T]... when ... -> ..., [... | mapfoo(T, ...)].
```

The base case can be branched by using guards as well. The parameters could be shuffled. In the most cases, either some body expressions or the head of list constructor refers to `H`.

### FARM11

```plaintext
map3(_, _, []) -> [];
map3(F, N, [H | T]) -> [F(H, N) | map3(F, N + 1, T)].
map3(F, 10, List)

[F(X, M) || {X, M} <- zip(List, seq(10, 10+length(List)))]

map3(_, [], []) -> [];
map3(N, [], [T]) -> [N | map3(N + 1, T)].
skelfarm(fun({X, M}) -> F(X, M) end,
  zip(List, map3(10, List)))
```
2.1.2 Pipeline

2.1.2.1 Static pipeline

Static pipelines are built up from a fixed number of stages. In contrast to dynamic pipelines, the number of stages is practically determinable in compile-time.

Syntactically, we are looking for map-like patterns (list comprehensions, maps, recursive functions) that apparently perform multiple computation steps on every element of a data list. In the most cases, these syntactic patterns are specialisations of those given in the case of the farm pattern; we will use the same categorisation.

Let us define the following two auxiliary functions:

```
skelpipe(FL, DL) ->
    skel:do([{pipe, [{seq, Fun} || Fun <- FL}]], DL).
funcomp(F, G) -> fun(X) -> F(G(X)) end.
```
### List comprehensions

<table>
<thead>
<tr>
<th>PIPE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>[foo(bar(X))</td>
</tr>
<tr>
<td>[(fun foo/1)(bar(X))</td>
</tr>
<tr>
<td>[(funcomp(fun foo/1, fun bar/1))(X)</td>
</tr>
<tr>
<td>skelpipe([fun bar/1, fun foo/1], List)</td>
</tr>
</tbody>
</table>

### Higher-order functions on lists

#### PIPE2: Applications of lists:map/2

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lists:map(fun(X) -&gt; foo(bar(X)) end, List)</td>
</tr>
<tr>
<td>lists:map(fun(X) -&gt; (fun foo/1)(bar(X)) end, List)</td>
</tr>
<tr>
<td>lists:map(funcomp(fun foo/1, fun bar/1), List)</td>
</tr>
<tr>
<td>skelpipe([fun bar/1, fun foo/1], List)</td>
</tr>
</tbody>
</table>

#### PIPE3: Applications of lists:map/2

*F and G should be 1-parameter funs*

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lists:map(fun(X) -&gt; F(G(X)) end, List)</td>
</tr>
<tr>
<td>lists:map(funcomp(F, G), List)</td>
</tr>
<tr>
<td>skelpipe([G, F], L)</td>
</tr>
</tbody>
</table>

#### PIPE4: Applications of lists:map/2 (more complex fun)

*Similar exemplars can be composed by using comprehensions instead of lists:map.*

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lists:map(fun(X) -&gt; 1 + F(G(X + 2) / 3) end, List)</td>
</tr>
<tr>
<td>skelpipe([fun(X) -&gt; G(X + 2) end, fun(X) -&gt; 1 + F(X / 3) end], L)</td>
</tr>
</tbody>
</table>
f(X) ->
    Y = G(X * 2),
    Z = F(Y / 3),
    g(Z).
g(Z) ->
    Z + 1.
lists:map(fun f/1, List)

skelpipe([fun(X) -> X * 2 end,
      fun(X) -> G(X) end,
      fun(X) -> X / 3 end,
      fun(X) -> F(X) end,
      fun(X) -> X + 1 end], L)

(Tail-recursive) Implementations of map

PIPE5 : Recursive functions

pipeline([]) -> [];
pipeline([H | T]) -> [foo(bar(H)) | pipeline(T)].
pipeline(L) -> skelpipe([fun bar/1, fun foo/1], L).

PIPE6 : Recursive functions (funs as variables)

pipeline(_, _, []) -> [];
pipeline(F, G, [H | T]) -> [G(F(H)) | pipeline(F, G, T)].
pipeline(F, G, L) -> skelpipe([F, G], L).

2.1.2.2 Dynamic pipeline

Dynamic pipelines are built up by instantiating the same stage(s) various times. The cardinality of the stages can only be determined in run-time.

In homogeneous dynamic pipelines, the same function is applied to the data list \( N \) times (in other words, we apply the \( N \)-times composition of the function with itself to every data element). Syntactically, there is, in addition to the list to be processed, a parameter \( N \) determining the number of iterations, consequently, the length of the pipeline whose stages will be identical.
Heterogeneous dynamic pipeline: multiple stages repeated \( N \) times. The transformation of such code can rely on the existing mechanism of selecting and executing the different stages.
2.1.3 Map

A list is turned into a list of lists, the original elements are decomposed. The resulting items are all processed one by one, and then the nested lists are recomposed each.

Such control and data flow can easily be implemented by list comprehensions, maps as well as with recursive functions, just like seen in the case of farms. Ideally, we will be able to identify any of these syntactic forms.

2.1.4 Reduce

The input list is aggregated into a single value by repeatedly applying the given operator (binary function). The pattern definition specifies that the operator be
associative and commutative, which is in turn a side condition to be checked in subsequent phases.

The implementation of the reduce pattern allows to apply the same reduce operation on several lists in parallel, and we also have an option to give a preparatory function applied on each input collection. Nevertheless, we only collected examples corresponding to the original description: reduce on a single list without any preparatory step. We define the following auxiliary functions in order to ease applying the skeleton implementation.

```erlang
id(X) -> X.
skelreduce(Fun, List) ->
    skel:do([[reduce, Fun, fun id/1]], [List]).

REDUCE1 : Fold with accumulator

The fold function could process the list elements in arbitrary order, because the reduction operator is associative and commutative.

```

```erlang
fold(Fun, Acc, List) ->
    lists:foldl(Fun, Acc, List).
fold(Fun, Acc, List) ->
    lists:foldr(Fun, Acc, List).
fold(Fun, Acc, []) -> Acc;
fold(Fun, Acc, [X | L]) ->
    fold(Fun, Fun(Acc, X), L).
fold(Fun, Acc, List) ->
    skelreduce(Fun, [Acc | List]).
```

```
REDUCE2 : Fold without accumulator

The lists module does not offer a fold operator without an accumulator, but we can apply the foldl function so that we choose an arbitrary item from the given list as accumulator.

```
fold(Fun, List) ->
    lists:foldl(Fun, hd(List), tl(List)).
fold(Fun, List) ->
    lists:foldr(Fun, hd(List), tl(List)).
fold(Fun, [X]) -> X;
fold(Fun, [X1, X2 | L]) ->
    fold(Fun, [Fun(X1,X2) | L]).
fold(Fun, List) ->
    skelreduce(Fun, List).
```
Higher-order functions on lists

**REDUCE3 : Application of lists:sum/1**

\[
\text{lists:sum(List)}
\]

\[
\text{skelreduce(fun (X,Y) \to X + Y end, List)}
\]

**REDUCE4 : Application of lists:min/1**

\[
\text{lists:min(List)}
\]

\[
\text{Fun = fun(X,Y) \to if X < Y \to X; true \to Y end, skelreduce(Fun, List)}
\]

Similarly, we can parallelize the application of lists:max/1 by changing the less-than operator to a greater-than operator in the parallel code fragment.

**REDUCE05 : Application of lists:all/1**

\[
\text{Note that the proposed transformation is only applicable if the input list does not contain boolean values.}
\]

\[
\text{lists:all(Pred, List)}
\]

\[
\text{Fun = fun (X,Y) \to X2 = if is_boolean(X) \to X; true \to Pred(X) end, Y2 = if is_boolean(Y) \to Y; true \to Pred(Y) end, X2 andalso Y2 end, skelreduce(Fun, List)}
\]

Similarly, we can parallelize the application of lists:any/1 by changing the andalso operator to orelse in the parallel code fragment.

### 2.2 High-level and Application-specific Patterns

In this section, we show some illustrative examples to the patterns defined in D2.3. These patterns are high-level, which means they are built up by using the core pat-
terns as building blocks. The materialisation of these patterns can greatly diverge, thus we try to present exemplars that highlight the main characteristics.

Throughout this section, \texttt{sk\_hlp} module denotes a module that contains the implementation of each high-level pattern, introduced in D2.6 [7].

### 2.2.1 Divide and Conquer

A function can be transformed to an instance of this pattern if it processes its list-typed parameter in the following way: initially it calls a condition function with the list, then according to its return value the d&c function simply returns with the value calculating by the so-called “base case solver” function on the list; or divides the list into smaller sub-lists (“divide” function), calls the function recursively on each sub-list and finally recomposes the processed sub-lists into a single list (“conquer” function).

The implementation of the pattern in \texttt{sk\_hlp} module takes the above mentioned four functions as parameters and returns with a parallel divide and conquer function.

\begin{verbatim}
DC1

divide_and_conquer(L) ->
  case is_base_case(L) of
    true -> solve_base_case(L);
    false ->
      LL = divide(L),
      ProcessedLL = [divide_and_conquer(L0) || L0 <- LL],
      conquer(ProcessedLL)
  end.

divide_and_conquer(L) ->
  case is_base_case(L) of
    true ->
      solve_base_case(L);
    false ->
      conquer(lists:map(fun divide_and_conquer/1,
                      divide(L)))
  end.

divide_and_conquer(L) ->
  Fun = sk_hlp:dc(fun is_base_case/1,
                 fun solve_base_case/1,
                 fun divide/1,
                 fun conquer/1),
  Fun(L).
\end{verbatim}

Besides this two sequential occurrences of the d&c algorithm it can vary syntac-
tically: for example, function calls in the “false” clause can be nested in different depths and new variables for the partial results can be introduced in different ways.

2.2.2 Pool Evolution

In D2.3, a very similar pattern was introduced, called “The Global Single Population Genetic pattern”, but as of D2.5, the pattern is mentioned as “Pool Evolution”. Also, the implementation (sk_hlp) of the high-level pattern set contains the pool evolution algorithm. Similarly to the d&c pattern implementation, it takes four functions as parameters and returns with a parallel pool evolution function.

As in case of d&c algorithm, the pool evolution algorithm has also many syntactically different occurrences. For example, the application of the “evolve” function on each selected values could be any map-like pattern, or instead of calling a user-defined filter function one can use the lists:filter/2 library function.
2.2.3 Orbit

Here, we show an algorithm that calculates the closure set of the given initial set as an exemplar to the Orbit pattern. As this pattern is not implemented yet, we show a possible realisation that uses the already present low-level skeletons.

**Orbit: Calculating closure**

```
general_closure(WorkerFun, InitialItems)->
general_closure00(WorkerFun,
    ordsets:from_list(InitialItems),
    ordsets:new()).

general_closure00(_, [], Items)->
    Items;
general_closure00(WorkerFun, UnprocedItems0, Items0)->
    UnprocedIs =
        ordsets:from_list(WorkerFun(UnprocedItems0)),
    Items = ordsets:union(UnprocedIs, Items0),
    general_closure00(WorkerFun,
        ordsets:subtract(UnprocedIs, Items0),
        Items).

general_closure_par(WorkerFun, InitialItems)->
    Constraint = fun({UnprocedISet, _ISet})->
        ordsets:size(UnprocedISet) /= 0
    end,
    OneStep = fun({UnprocedItems0, Items0})->
        NumOfWorkers = 2,
        FarmFun = fun(Item)-> WorkerFun([Item]) end,
        Farm = {farm,[[seq, FarmFun]], NumOfWorkers},
        FarmRes = skel:do([Farm],
            ordsets:to_list(UnprocedItems0)),
        UnprocedItems =
            ordsets:from_list(lists:flatten(FarmRes)),
        Items = ordsets:union(UnprocedItems, Items0),
        (ordsets:subtract(UnprocedItems, Items0), Items)
    end,
    Feedback = [feedback, [{seq, OneStep}], Constraint],
    Init0 = [{ordsets:from_list(InitialItems),
        ordsets:new()}],
    [[_, Unproced, Items]] = skel:do([Feedback], Init0),
    Items.
```

2.2.4 Possibly Iterated Numerical Library Calls

Calculating the closest primes to the given numbers can be an example for the Possible Iterated Numeric Library Calls pattern.
### Possibly Iterated Numerical Library Calls: Determining next primes

<table>
<thead>
<tr>
<th>next_primes(Nums) -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>[next_prime(Num)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next_prime(Num) -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>case is_prime(Num) of</td>
</tr>
<tr>
<td>true -&gt; Num;</td>
</tr>
<tr>
<td>false -&gt; next_prime(Num+1)</td>
</tr>
<tr>
<td>end.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>is_prime(1) -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>false;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>is_prime(N) when N&lt;4 -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>true;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>is_prime(Num) -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>lists:all(fun(I)-&gt; Num rem I /= 0 end,</td>
</tr>
<tr>
<td>lists:seq(2,trunc(Num/2))].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next_primes(Nums) -&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint = fun(N)-&gt; not is_prime(N) end,</td>
</tr>
<tr>
<td>NumOfWorkers = 2,</td>
</tr>
<tr>
<td>Farm = {farm,[{seq, fun(X)-&gt; X+1 end}], NumOfWorkers},</td>
</tr>
<tr>
<td>skel:do([{{feedback, [Farm], Constraint}]], Nums).</td>
</tr>
</tbody>
</table>

### 2.2.5 (Colored) Stencil

An object lesson for the Stencil pattern can be the *Game of Life* [2] that is a cellular automaton, in which the next state of each cell depends on the following rules.

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The shown Erlang source implements one step of this game.
Stencil01: Game of Life

```prolog
\[
\text{cell_neighbours}([X,Y]) ->
  \{(X,Y),
   \[(I,J) \mid I \leftarrow [X-1, X, X+1], J \leftarrow [Y-1, Y, Y+1],
     I>0, I \leq N, J>0, J \leq N, \{I\neq X \text{ or else } J\neq Y\}\}.\]
\]

\[
\text{one_step_of_game_of_life}(\text{InitialLiveCells}) ->
  \text{Table} = \{[I,J] \mid I \leftarrow \text{lists:seq}(1,N),
                 J \leftarrow \text{lists:seq}(1,N)\},
  \text{Neighbours} = \text{lists:map}(\text{fun} \text{cell_neighbours/1}, \text{Table}),
  \text{Interaction} = \text{fun}(\text{fun}([C, \text{Nhs}]) ->
    (\text{C}, [\text{case} \text{lists:member}(\text{Cell}, \text{InitialLiveCells}) \text{ of}
                \text{true} \to 1;
                \text{false} \to 0
                \text{end} \mid \text{Cell} \leftarrow \text{Nhs}])
  \end,\]
  \text{InteractedNhs} = \text{lists:map}(\text{Interaction}, \text{Neighbours}),
  \text{Effect} = \text{fun}(\text{fun}([C, \text{INhs}]) ->
    \text{LiveNhsCount} = \text{lists:sum}(\text{INhs}),
    \text{CellAlive} = \text{lists:member}(\text{C}, \text{InitialLiveCells}),
    \text{case} \text{CellAlive} \text{ of}
    \text{true when LiveNhsCount < 2} \to \{\text{C, dead}\};
    \text{true when LiveNhsCount < 4} \to \{\text{C, alive}\};
    \text{true} \to \{\text{C, dead}\};
    \text{false when LiveNhsCount == 3} \to \{\text{C, alive}\};
    _ \to \{\text{C, dead}\}
    \text{end},\]
  \text{lists:map}(\text{Effect}, \text{InteractedNhs}).\]
\]
```

2.3 Patterns in Parallel Programs

In this section, we show some exemplars to some additional categories, so this section represents a kind of "escape" from the general ParaPhrase methodology, as these categories have never been considered in the other WP2 deliverables.

Due to the built-in support for concurrency and distribution, it is likely to happen that the analysed Erlang applications have been partially parallelised at least. Thus, exemplars that highlight the common properties of these concurrent structures are necessary.

In most cases, the parallelised parts do not scale well, and can also be identified and transformed by introducing Skel library calls that eliminate hand-made parallelisms. By transforming these patterns, applications can achieve significant speed-ups and also become more scalable.
2.3.1  RPC Calls

Erlang/OTP contains a module, called \( rpc \) \[1\], that provides remote procedure call services and even more: parallel evaluators, broadcast facilities and utilities to collect information from a remote node.

### RPC01: RPC Calls

| worker(List, Op) when is_list(List) ->          |
| [worker(Item, Op) || Item <- List];          |
| worker(Item, add) ->                         |
| Item + 1;                                    |
| worker(Item, sub) ->                         |
| Item - 1.                                    |

```erlang
map(List, Op) ->
    FunToBeApplied = {?MODULE, worker},
    rpc:pmap(FunToBeApplied, [Op], List).
```

```erlang
map2(ListofList, Op) ->
    TKeys =
        [rpc:async_call(node(), ?MODULE, worker,
            [SubList, Op])
            || SubList <- ListofList],
    [rpc:yield(K) || K <- TKeys].
```

```erlang
map(List, Op) ->
    Fun = fun(Args) ->
        apply(?MODULE, worker, Args)
    end,
    Pre = fun(ListElem) -> [[ListElem, Op]] end,
    Post = fun([X]) -> X end,
    InnerWF1 = [{ord, [{farm, [{seq, Fun}], length(List)}]}],
    skel:do([{cluster, InnerWF1, Pre, Post}], List).
```

2.3.2  Hand-made parallelism

Erlang provides built-in primitives to manage both processes and inner-process communications. New processes can be started by calling one of the `spawn` variants encapsulated by the `erlang` module. Messages can be sent to an Erlang process either by applying the `!` operator or by calling one of the `send` variants located in the `erlang` module. To receive a message from an Erlang process, the `receive` language element should be used.

Due to the simplified process handling, programmers can easily overuse these primitives, which can cause a performance bottleneck and can make the whole application become a non-scalable application. Thus, we have to deal with these constructs.
2.3.3 Hand-made skeletons

It is likely to happen that programmers have made their own library to handle and manage concurrent program parts uniformly. It is noticeable that Skel and these libraries stand for the same purpose and have some common parts. If these libraries can be identified and transformed, every program parts depending on them can achieve significant speed-up. Additionally, the whole application becomes more scalable, thus discovering these patterns are desired.
Hand-made skeleton

```erlang
farm(List, Module, Function, MakeProperArgs) when is_list(List) andalso is_atom(Module) andalso is_atom(Function) andalso is_function(MakeProperArgs, 1) ->
    Lists = util:partition_list_according_cpus(List),
    Keys = [rpc:async_call(node(), Module, Function, MakeProperArgs(SubList)) || SubList <- Lists],
    Res = lists:append([rpc:yield(K) || K <- Keys]),
    lists:zip(List, Res).
```

```erlang
farm(List, Module, Function, MakeProperArgs) when is_list(List) andalso is_atom(Module) andalso is_atom(Function) andalso is_function(MakeProperArgs, 1) ->
    Fun = fun(Args) ->
        apply(Module, Function, MakeProperArgs(Args))
    end,
    NumOfWorkers = 2,
    Pre = fun(X) -> [[X]] end,
    Post = fun([[X]]) -> X end,
    InnerWFl = [{ord,[[farm,[seq, Fun]], NumOfWorkers]]},
    Res = skel:do([{cluster, InnerWFl, Pre, Post}], List),
    lists:zip(List, Res).
```

### 2.3.4 Distributed applications

Although ParaPhrase project focuses only on parallelism, Erlang provides built-in support for distribution in a transparent manner that is heavily used by real-world Erlang applications. There are different causes that make programmers construct distributed systems to solve problems. The main cases are listed as follows.

- All of the data used by the application is not globally available to each Erlang node.

- A better performance can be achieved by distributing the necessary computations among Erlang nodes running on separate machines.

The latter case is only important for us, because the same problem may be solved more efficiently by a parallel system that utilises ParaPhrase’s technologies. In spite of the expected advantages, the re-design of distributed applications is a hard, time-consuming work that cannot be done in every case.
Bibliography


