Using Program Shaping to Parallelise an Erlang Multi-Agent System

Adam D. Barwell, Chris Brown, Kevin Hammond
University of St Andrews

Aleksander Byrski, Wojciech Turek
AGH University of Science and Technology

Lambda Days 2016, 18 February 2016
RePhrase Project: Refactoring Parallel Heterogeneous Software – a Software Engineering Approach (ICT-644235), 2015-2018, €3.6M budget

8 Partners, 6 European countries
UK, Spain, Italy, Austria, Hungary, Israel

Coordinated by St Andrews
What are we trying to achieve?

Parallelism and Concurrency
Key Software Engineering Challenges

- Testing, Verification and Debugging
  - Automatic Test Generation, race condition detection, ...

- Software Quality Assurance
  - New Standards are needed
  - Cross-Platform Approaches

- Deployment on heterogeneous platforms
  - e.g. CPU/GPU, APU, manycore, FPGA
  - efficient scheduling of multiple applications

- Maintainability and Software Evolution
  - Change parallelism structure
  - Adapt to varying numbers of cores and processor types
## The RePhrase Approach

<table>
<thead>
<tr>
<th>Software development phase</th>
<th>RePhrase tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements capture</td>
<td>Parallel requirements capturing methods</td>
</tr>
<tr>
<td>Design</td>
<td>Refactoring tool, patterns</td>
</tr>
<tr>
<td>Implementation and debugging</td>
<td>Refactoring tool, pattern implementations, adaptivity tool</td>
</tr>
<tr>
<td>Testing and verification</td>
<td>Parallel testing framework, parallel verification tool, failure detection tool, property violation detection tool</td>
</tr>
<tr>
<td>Deployment</td>
<td>Adaptivity tools, refactoring tool</td>
</tr>
<tr>
<td>Maintenance and evolution</td>
<td>Refactoring tool, adaptivity tools, patterns, pattern implementations, quality assurance tool</td>
</tr>
</tbody>
</table>
Program Shaping

- Restructuring (legacy) programs to enable the introduction of parallelism
- Might include:
  - Removing certain types of side effects
  - Encapsulating computations into components
  - Eliminating unnecessary dependencies
- Currently *ad hoc*
  - Often non-trivial
  - Requires intimate knowledge of code, language, and parallelism
- Refactoring techniques can be used to automate the process
Refactoring for Parallelism

- Conditional, source-to-source transformation that preserves functional correctness
- Refactoring tools to help automate this process
  - Semi-automatic transformation avoids introducing errors
  - Developer input allows a wider range of possible transformations
- **Wrangler**
  - Extensible refactoring tool for Erlang
  - Built-in collection of refactorings
  - API allows user creation of refactorings
  - Originally created by the University of Kent
Algorithmic Skeletons

• High-level abstraction of some common pattern of parallelism

• Composable and nestable

• Language independent

• Need only problem-specific sequential code
  • plus any skeletal parameters

• Implemented and collected in *algorithmic skeleton libraries*
Example Skeletons

Pipeline

<table>
<thead>
<tr>
<th>x₀</th>
<th>x₁</th>
<th>x₂</th>
<th>x₃</th>
<th>x₄</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Inputs

Stage 1

Stage 2

Stage 3

...  

Stage n

Outputs

<table>
<thead>
<tr>
<th>x₀'</th>
<th>x₁'</th>
<th>x₂'</th>
<th>x₃'</th>
<th>x₄'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Task Farm

<table>
<thead>
<tr>
<th>x₀</th>
<th>x₁</th>
<th>x₂</th>
<th>x₃</th>
<th>x₄</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Inputs

Emitter

Workers

Collector

Outputs

<table>
<thead>
<tr>
<th>x₀'</th>
<th>x₁'</th>
<th>x₂'</th>
<th>x₃'</th>
<th>x₄'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Skeletons

Feedback

Inputs
\[ x_0 \ x_1 \ x_2 \ x_3 \ x_4 \]

Inner-workflow

Satisfies Condition? 

Yes

No

Outputs
\[ x'_0 \ x'_1 \ x'_2 \ x'_3 \ x'_4 \]
Skel

• An algorithmic skeleton library for Erlang
  • Pipeline, Task Farm, Feedback, and more
  • Hybrid skeletons for both CPU and GPU targets
  • http://skel.weebly.com
Skel Example

```
lists:map(fun worker/1, Input)
```

Introduce Farm

```
skel:do({farm, fun worker/1, NW}, Input)
```
Multi-Agent Systems

- An agent is an intelligent, autonomous entity that solves some problem or subtask.

- A Multi-Agent System (MAS) brings two or more agents together to solve some complex problem, e.g. flood forecasting.

- An Evolutionary Multi-Agent System (EMAS) combines multi-agent systems with evolutionary algorithms.

- Highly parallel: agents are independent.
Evolutionary Multi-Agent Systems (EMAS)

- Meta-heuristic approach for optimization
  - universal optimization algorithm (formally proven)
- Explicit hybridisation of agent-oriented and evolutionary computing
- Agents
  - contain genotypes and energy as a means for distributed selection
  - located on evolutionary islands
  - perform actions (death, reproduction, migration, fight)
EMAS – Basic Structure

- Tag
- Migrant
- Group
- Shuffle
- Agents

The diagram shows the relationships between these components, with arrows indicating the flow or interaction between them.
EMAS Code

loop(Islands, Time, SP, Cf) ->
  Tag = fun(Island) ->
    [{
      mas_mis util: behaviour proxy(
        Agent, SP, Cf),
      Agent} || Agent <- Island]
    end,
  Groups = [mas_mis util: group by(Tag(I)) || I <- Islands],
  Migrants = [seq_migrate(lists:keyfind(migration, 1, Island), Nr) || {Island, Nr} <-
    lists:zip(Groups,
    lists:seq(1,
    length(Groups)))],
  NewGroups = [[mas_mis util: meeting proxy(
    Activity,
    mas_sequential,
    SP,
    Cf) || Activity <- I] || I <- Groups],
  WithMigrants = append(
    lists:flatten(Migrants),
    NewGroups),
  NewIslands = [mas_mis util: shuffle(lists:flatten(I)) || I <- WithMigrants],
  case os:timestamp() < Time of
    true ->
      loop(NewIslands, Time, SP, Cf);
    false ->
      NewIslands
  end.
Parallelising EMAS

• We could introduce a task farm for each list comprehension...

• But this is inefficient:
  • Farm creation creates overhead
  • Not all tasks in the system are large enough for parallelism
  • The function loops until some condition is met, compounding the above issues

• It is better to express the parallel behaviour in a single operation
  • Knowing the full structure allows minimisation of overheads

• However, as the code stands, we cannot introduce this operation
Program Shaping Refactorings

• First divide up the sequential code into atomic *components*
  • which we can then rearrange

• We will use the following refactorings:
  • Extract Composition Function
  • Compose Maps
  • Intro Func
  • Intro Farm
  • Intro Feedback
  • Intro Skel
First, encapsulate code for stages into blocks using Extract Composition Function

```
TagFun =
    fun (Agent) ->
        {mas_misc_util:behaviour_proxy(Agent, SP, Cf),}
    end,
Tagged = lists:map(TagFun, Islands),
```
Stage 1

Split and format the tagging, grouping, and migrating stages into components using Extract Composition Function.

TagFun =
  fun (Agent) ->
    {mas_misc_util:behaviour_proxy(Agent, SP, Cf),
     Agent}
  end,
Tagged = lists:map(TagFun, Islands),
GroupFun = fun (I) -> mas_misc_util:group_by(I) end,
Groups = lists:map(GroupFun, Tagged),
MigrantFun =
  fun ({Island, Nr}) ->
    seq_migrate(lists:keyfind(migration, 1, Island), Nr)
  end,
Migrants = lists:map(MigrantFun,
                    lists:zip(Groups,
                              lists:seq(
                                           1,
                                           length(Groups)))),
Stage 2

Since these stages can be easily composed, using the classical Inline Method refactoring, we inline the migration function.

```haskell
MigrantFun =
  fun ({{migration, Agents}, From}) ->
    Destinations =
      [{mas_topology:getDestination(From),
        Agent} || Agent <- Agents],
    mas_misc_util:group_by(Destinations);
    (OtherAgent) -> OtherAgent
  end,
Migrants = lists:map(MigrantFun,
    lists:zip(Groups,
           lists:seq(1,
                     length(Groups)))))
```
EMAS – Program Shaping

Next, group together stages and remove dependencies using Compose Maps
Stage 3

We now compose the tagging, grouping, and migrating stages using Compose Maps.

```plaintext
TagFun = fun (Agent) ->
    {mas_misc_util:behaviour_proxy(Agent, SP, Cf),
     Agent}
end,

GroupFun = fun (I) -> mas_misc_util:group_by(I) end,

MigrantFun =
    fun ({{migration, Agents}, From}) ->
        Destinations =
            [{mas_topology:getDestination(From), Agent} || Agent <- Agents],
        mas_misc_util:group_by(Destinations);
        (OtherAgent) -> OtherAgent
    end,

TGM = fun(Agents) ->
    Tagged = lists:map(TagFun, Agents),
    Migrants = lists:map(MigrantFun, Tagged),
    GroupFun(Migrants)
end,

TGMs = lists:map(TGM, Islands),
```
Stage 4

We expose functions as components for new groups and new islands stages using Extract Composition Function

```
NewGroupsFunInnerFun =
  fun (Activity) ->
    mas_misc_util:meeting_proxy(Activity, mas_sequential, SP, Cf)
  end,
NewGroupsFun =
  fun (I) ->
    lists:map(NewGroupsFunInnerFun, I)
  end,
NewGroups = lists:map(NewGroupsFun, TGMs),

NewIslandsFun =
  fun (I) ->
    mas_misc_util:shuffle(lists:flatten(I))
  end,
NewIslands = lists:map(NewIslandsFun, NewGroups),
```
Next, create a farm of agents using Intro Farm
Stage 5

We now start to arrange these individual components, ready to be passed to Skel. We apply Intro Func over TGM and NewGroupsInnerFun expressions. We also introduce a farm over NewGroupsFun

```plaintext
TGMs = {func, TGM},
Work =
  {func,
   fun (Activity) ->
   mas_misc_util:meeting_proxy(Activity,
   mas_farm,
   SP,
   Cf)
  end},
Map = {farm, [Work], Cf#config.skel_workers},
NewGroups = lists:map(NewGroupsFun, TGMs),
Shuffle =
  fun (I) ->
  mas_misc_util:shuffle(lists:flatten(I))
  end,
NewIslands = lists:map(Shuffle, NewGroups),
```
Stage 6

We use Intro Func on Shuffle, completing the skeletons needed for Skel, and use Intro Skel over NewIslands and NewGroups

```
Shuffle =
    {func,
     fun (I) ->
         mas_misc_util:shuffle(lists:flatten(I))
     end},

Pipe = {pipe, [TGMs, Map, Shuffle]},
NewIslands =
    [NewIsland ||
     {_, NewIsland} <- skel:do([Pipe], Islands)],
```
EMAS – Program Shaping

Finally, use Intro Feedback using the pipeline and farm as components
Stage 7 (End)

We use Intro Feedback to fold the outer loop into the Skel invocation, improving efficiency.

This completes the shaping and parallelisation process.
EMAS Code (Shaped)

```haskell
loop(Islands, Time, SP, Cf) ->
    EndTime =
        mas_misc_util:add_miliseconds(os:timestamp(), Time),

    TagFun =
        fun (Agent) ->
            {mas_misc_util:behaviour_proxy(Agent,
                SP,
                Cf), Agent}
        end,

    GroupFun = fun (I) -> mas_misc_util:group_by(I) end,

    MigrantFun =
        fun ([{migration, Agents}, From}) ->
            Destinations =
                [{mas_topology:getDestination(From),
                Agent} || Agent <- Agents],
            mas_misc_util:group_by(Destinations);
            (OtherAgent) -> OtherAgent
        end,

    TGM = tgm(TagFun, GroupFun, MigrantFun),
    TGMS = {func, TGM},

    Work = {func,
        fun (Activity) ->
            mas_misc_util:meeting_proxy(
                Activity,
                mas_farm,
                SP,
                Cf)
        end},

    Map = {farm, [Work], Cf#config.skel_workers},

    Shuffle = {func,
        fun (I) ->
            mas_misc_util:shuffle(lists:flatten(I))
        end},

    Pipe = {pipe, [TGMS, Map, Shuffle]},
    Constraint = fun (_) -> os:timestamp() < Time end,
    FinalIslands = skel:do([{farm, [{feedback, [Pipe], Constraint}],
               Cf#config.skel_workers}],
               [Islands]).
```
Results

• We compare our shaped EMAS to two other versions:
  1. **Concurrent**: follows good Erlang practice for writing concurrent code;
  2. **Hybrid**: designed and manually tuned to give the best possible performance for the EMAS algorithm

• Two different benchmarks
  • continuous (**Rastrigin**)
  • discrete (**Low Autocorrelation Binary Sequences**)

• Tested on a 64-core machine at AGH, Poland (ZEUS)
  • 4 x AMD Opteron 6276, 16 2.3GHz cores
Optimization Benchmark

- Find optimum of Rastrigin function in dimensions $n = 100$
  - $f(x) = 10n + \sum_{i=1}^{n}(x_i^2 - 10 \cos(2\pi x_i))$
  - One of classic global optimization benchmark functions

- Example: Rastrigin function in two dimensions
**LABS**

**Low-Autocorrelation Binary Sequences**

- $S = s_1 s_2 \ldots s_L$: binary sequence of length $L$ and $s_i \in \{-1, +1\}$
- Aperiodic Autocorrelation with lag $k$: $C_k(S) = \sum_{i=1}^{L-k} s_i s_{i+k}$
- Minimize $E(S) = \sum_{k=1}^{L-1} C_k^2(S)$ with respect to $S$

\[ S = s_1 s_2 \ldots s_L : \text{binary sequence of length } L \text{ and } s_i \in \{-1, +1\} \]

\[ A \text{periodic Autocorrelation with lag } k : \]

\[ C_k(S) = \sum_{i=1}^{L-k} s_i s_{i+k} \]

\[ \text{Minimize } E(S) = \sum_{k=1}^{L-1} C_k^2(S) \text{ with respect to } S \]

Find $S$
Speedups for Rastigin Function

- Skel
- Hybrid
- Concurrent
Speedups for LABS

Graph showing speedups for different core counts.

- Skel
- Hybrid
- Concurrent

Cores range from 1 to 64.

Speedup range from 0 to 80.
**EMAS : Coding Efficiency**

- Effort for implementing the generic EMAS backends

<table>
<thead>
<tr>
<th>Method</th>
<th>Lines of Code</th>
<th>Effort in Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>85</td>
<td>10</td>
</tr>
<tr>
<td>Hybrid</td>
<td>129</td>
<td>2</td>
</tr>
<tr>
<td>Concurrent</td>
<td>353</td>
<td>7</td>
</tr>
<tr>
<td>SKEL</td>
<td><strong>100</strong></td>
<td><strong>1</strong></td>
</tr>
</tbody>
</table>
Conclusions

• We have introduced **novel program shaping techniques**
  • applied to an Erlang implementation of an Evolutionary Multi-Agent System, a real-world use case

• Obtain speedups of 45x for *Rastrigrin* and 70x for *LABS*
  • *at minimal programmer effort*

• Applicable to other languages, e.g. C++, Java, ...
Future Work

- Other use cases, and further evaluate the effectiveness of the approach; e.g. the Dialyzer

- Expansion of our library of program shaping techniques

- Incorporate static analysis techniques to further automate the program shaping process, at the same time reducing the burden on the programmer

- Demonstrate the applicability of this approach to use cases in languages other than Erlang
THANK YOU!

http://rephrase-ict.eu

@rephrase_eu