Parallelising an Erlang Multi-Agent System

Adam D. Barwell Kevin Hammond
University of St Andrews
{adb23, kh8}@st-andrews.ac.uk

Aleksander Byrski Wojciech Turek
AGH, Krakow
olekb@agh.edu.pl, wojciech.turek@agh.edu.pl

Abstract

Program Shaping is the process of transforming a sequential program in order to better enable the introduction of parallelism. While *algorithmic skeletons* abstract away the low-level aspects of parallel programming that often plague traditional techniques, skeletons cannot always be readily introduced to sequential code. Data may not always be in a compatible format, function design may not be conducive to a single point of invocation, or there may be dependencies between functions and data obstructive to the introduction of parallelism. Program Shaping can be used to transform such code, producing a form to which skeletons can be introduced. We present a series of generic Program Shaping rewrite rules, and their implementation as refactorings, and demonstrate their application to an Erlang Multi-Agent System (MAS).

Keywords Program Shaping, Refactoring, Parallelism, Algorithmic Skeletons, Multi-Agent Systems

1. Introduction

Once the preserve of specialist hardware, parallel-capable processors can now be found in devices common to everyday life [32]. This increased relevance has rendered parallel programming a necessary skill for the average programmer. Traditional parallelisation techniques consist of low-level primitives and libraries that require the programmer to manually introduce and manage the components of parallelism, e.g. threads, communication, locking, and synchronisation. This results in a process which is often tedious, difficult, and error-prone [16]. In response, recent years have seen a range of approaches designed to simplify the parallelisation process [9, 23, 24, 26, 33]. While these approaches differ, they each abstract away low-level parallel mechanics, a common source of error.

Although beneficial, these abstractions serve to address a specific problem. Other aspects of parallelism, such as which part(s) of a program should be parallelised [5] or which configuration gives best performance gains [1], must be similarly addressed. One common and non-trivial aspect of parallelisation, which heretofore has seen little interest, is the restructuring of code to enable the introduction of parallelism. Ranging from the detection and breaking of inter-task dependencies, through changing data representation to avoid excessive memory use, to the avoidance of scheduler inefficiencies, restructuring to enable parallelism requires extensive knowledge of the program code, the language used, and parallelism itself. The difficulty of this restructuring makes it a significant stage of the parallelisation process. Where we define *program shaping* to be *a series of intentional changes to source code that contribute towards some goal*, this restructuring stage is a form of program shaping. Presently a manual, *ad hoc*, and error-prone process, if we are to truly address the difficulty of parallel programming we must also address the difficulties presented by program shaping.

A *refactoring* [13] is a conditional, source-to-source program transformation that maintains functional correctness. While accomplishable manually, refactoring tools enact transformations both automatically and *safely*. Such tools exist for a wide range of languages, often featuring integration with popular editors [27]. Where recent work has demonstrated that refactoring can be used to automate the introduction of parallelism [4], it is possible to extend this approach to the program shaping stage.

An evolutionary multi-agent system is a hybrid meta-heuristic system designed to solve complex problems ranging from power systems management [25] to flood forecasting [14]. These systems divide their task between autonomous interacting agents, with each focussing on a particular subtask. In an evolutionary multi-agent system these agents are split into populations, and their solutions iteratively improved using genetic algorithms. Whilst the independent nature of agents suggest these an ideal target for parallelisation, the complexity of the systems as a whole make this difficult.

In this paper we demonstrate how such an evolutionary multi-agent system built in Erlang by AGH might be semi-automatically restructured and parallelised using program shaping refactoring techniques. We provide a description of the refactorings used below, and demonstrate that, in combination with the skeletons, good performance gains can be achieved.

2. Background

2.1 Algorithmic Skeletons

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction [9] into parameterised templates. There has been a long-standing connection between the skeletons community and the functional programming community. In the functional world, skeletons are effectively higher-order functions that can be instantiated with specific user code to give some concrete parallel behaviour. For example, we might define a *parallel map* skeleton, whose functionality is identical to a standard *map* function, but which creates a number of processes (*worker processes*) to execute each element of the map in parallel.

Using a skeleton approach allows the programmer to adopt a top-down *structured* approach to parallel programming, where skeletons are composed to give the overall parallel structure of the program. This gives a flexible semi-implicit approach, where the parallelism is exposed to the programmer only through the choice

HLPGPU '16, Jan 19, 2016, Prague, Czech Republic.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

of skeleton and perhaps some specific behavioural parameters (e.g. the number of parallel processes to be created, or how elements of the parallel list are to be grouped to reduce communication costs). Details such as communication, task creation, task or data migration, scheduling etc. are embedded within the skeleton implementation, which may be tailored to a specific architecture or class of architectures. This offers an improved level of portability over the typical low-level approaches. However, it will still be necessary to tune behavioural parameters in particular cases, and it may even be necessary to alter the parallel structure to deal with varying architectures (especially where an application targets different classes of architecture). A recent survey of algorithmic skeleton approaches can be found at [15].

2.1.1 Skel

The *Skel* [4] library defines a small set of classical skeletons for Erlang. Each skeleton operates over a stream of input values, producing a corresponding stream of results. Because each skeleton is defined as a streaming operation, they can be freely substituted provided they have equivalent types. The same property also allows simple composition and nesting of skeletons. This paper will consider the following subset of the *Skel* skeletons:

- `func` is a simple wrapper skeleton that encapsulates a function, $f : a \to b$, as a skeleton, enabling the use of the function within Skel.
- `pipe` models a parallel pipeline over a sequence of skeletons $s_1, s_2, \ldots, s_n$ as a skeleton, enabling parallel composition of skeletons.
- `farm` models a task farm with n workers, whose operation is the skeleton s.
- `feedback` models a feedback skeleton that allows inputs to be applied to some skeleton s repeatedly until some condition c is met.

2.2 Refactoring

Refactoring is the process of changing the internal structure of a program, while preserving its (functional) behaviour. In contrast to general program transformations, refactoring focuses on purely structural changes rather than on changes to program functionality, and is generally applied semi-automatically, i.e. under programmer direction. This allows programmer knowledge about i.e. safety properties to be exploited, and so permits a wider range of possible transformation. Refactoring has many advantages over traditional transformation and fully automated approaches, including (but not limited to):

- Refactoring is aimed at improving the design of software. As programmers change software to meet new requirements, so the code loses structure; regular refactoring helps tidy up the code to retain a good structure.
- Refactoring makes software easier to understand. Programmers often write software without considering future developers. Refactoring can enable the code to better communicate its purpose.
- Refactoring encourages code reuse by removing duplicated code [2].
- Refactoring helps the programmer to program faster and more effectively by encouraging good program design.
- Refactoring helps the programmer to reduce bugs. As refactorings typically generate code automatically, it is easy to guarantee that this code is safe and correct [31].

The term 'refactoring' was first introduced by Opdyke in his 1992 PhD thesis [28], but the concept goes back at least to Darlington and Burstall's 1977 *fold/unfold* transformation system [6], which aimed to improve code maintainability by transforming Algol-style recursive loops into a pattern-matching style commonly used today. Historically, most refactoring was performed manually with the help of text editor 'search and replace' facilities. However, in the last couple of decades, a diverse range of refactoring tools have become available for various programming languages, that aid the programmer by offering a selection of automatic refactorings. For example, the most recent release of IntelliJ IDEA refactorer supports 35 distinct refactorings for Java [11]. Typical refactorings include *variable renaming* (changes all instances of a variable that are in scope to a new name), *parameter adding* (introduces a new parameter to a function definition and updates all relevant calls to that function with a placeholder), *function extraction* (lifts a selected block of code into its own function) and *function inlining*.

2.2.1 Wrangler

Whilst the refactoring community has produced a great deal of work on refactoring for the object-oriented paradigm [10], the concept is nevertheless applicable to a wide range of programming styles and approaches. Functional programming is no exception to this. Indeed, Darlington and Burstall's transformation system for recursive functions produces code that would not be out of place in modern functional programs [6]. The University of Kent has since produced the HaRe [21] and Wrangler [22] refactoring tools for Haskell and Erlang respectively. Both tools are implemented in their respective languages, and offer a number of standard refactorings. Wrangler is implemented in Erlang and is integrated into both Emacs and Eclipse. We exploit a recent Wrangler extension which allows refactorings to be expressed as AST traversal strategies in terms of their pre-conditions and transformation rules. The extension comes in two parts: a user-level language for describing the refactorings themselves [20]; plus a Domain-Specific-Language to compose the refactorings [20].

2.3 Multi-Agent Systems

A common approach to problem solving is to decompose that problem into smaller tasks, solving each (sub)task individually and later combining their solutions to produce an overall solution. This approach to problem solving lends itself well to parallel and distributed computing, where subtasks can be run indepenently on separate cores or machines. A typical example of this approach is the master-slave evolution model [7]. Multi-agent systems extend this approach by treating the processes that solve the tasks as intelligent, autonomous agents, with each agent capable of interacting with their environment and other agents. As their name implies, a multi-agent system (MAS) combines two or more of these autonomous agents, making them ideally suited for representing probelms that have many solving methods, involve many perspectives, and/or may be solved by many entities [35]. One of the major application areas of multi-agent systems is large-scale computing [34].

Evolutionary multi-agent systems are a hybrid meta-heuristic, combining multi-agent systems with evolutionary algorithms. Under these systems, agents are split into subsets, *populations*, and agents evolve within each population to improve their ability to solve a particular optimisation problem. As agents are designed to be autonomous, meaning no global information between them, the perturbation and selection processes of evolutionary algorithms must be decentralised. Under evolutionary multi-agent systems, this perturbation and selection process is therefore done via peer-to-peer interactions between agents.

Whilst the problems solved by both evolutionary and standard multi-agent systems are vairied, the approach and design of under-

$$
\begin{array}{lcl}
\gamma & = & \text{Program Environment} \\
\mathbb{F} & = & \text{Set of all functions in } \gamma \\
\mathbb{L} & = & \text{Set of all lists in } \gamma \\
\mathbb{S} & = & \{skel, \bar{f}, pipe, farm\} \\
\Gamma & = & \gamma \cup \mathbb{S}
\end{array}
$$

Figure 1. Environment Definitions for Refactorings

lying systems is often standard. Recent work by AGH [18] have developed patterns to describe the operation of evolutionary and multi-agent systems, with an exemplar implementation in Erlang. Where this implementation was initially sequential, through the technqiues and methodology described below we have been able to successfully parallelise it.

3. Rewrite Rules

To facilitate program shaping for the use case and beyond, we define a series of refactorings, described below. All refactorings presented below can be described as semi-formal rewrite rules, operating over the abstract syntax tree (AST) of the source program. Each refactoring has a set of conditions ensuring the transformation valid, a description of the syntax to be transformed, and a description of the syntax following successful transformation. Conditions are given as predicates to each rule.

Each rewrite rule operates within an environment γ allowing access and reference to the current scope of the rewrite rule within the source program. This includes the set of all available functions $\mathbb{F}$. The skeleton library Skel, and the skeletons it provides are denoted by the set $\mathbb{S}$.

$$
\begin{array}{rcl}
\mathbb{S} & = & \{skel, \bar{f}, pipe, farm, farm', farm'', \\
 & & cluster, cluster', cluster''\}
\end{array}
$$

For all rewrite rules, $\mathbb{S}$ is assumed to be in scope. This denoted in each rule by extending γ:

$$\Gamma = \gamma \cup \mathbb{S}$$

We define a series of semantic equivalences to allow for more concise rewrite rules. Each equivalence is subject to a series of predicates under which it is valid, and is defined in the form:

$$\bar{s}, xs \in \Gamma, xs : list\ a \vdash skel(\bar{s}, xs) = \texttt{skel} : \texttt{do}(\bar{s}, xs)$$

Where $\bar{s}$ represents any valid skeleton in $\mathbb{S}$, i.e. $\mathbb{S}/\{skel\}$; and xs evaluates to a list where all elements have the same type. Semantic equivalences have been defined for each skeleton in Skel; with pertinent definitions given in Fig. 2.

Using these semantic equivalences we can define rewrite rules for each refactoring. Due to space limitations, we define two of our refactorings in this format, giving textual descriptions of the others in Fig 3.

3.1 Extract Composition Function

The *Extract Composition Function* refactoring exposes sequential functionality that may later be used as part of a parallel pipeline. Whilst it is possible, e.g. via *Intro Farm*, to immediately introduce a skeleton over a list comprehension, such refactorings commonly assume the list comprehension is the top-level command. Where the list comprehension is nested within a loop, or is just part of a solution, it can be advantageous to lift each part of the solution into atomic closures of sequential functionality which can later be arranged into an optimal configuration for parallelism.

$$
\begin{array}{lcl}
\Gamma & \vdash & \mathcal{F} \\
 & = & F \\
 & | & \texttt{fun ?MODULE :} f/1 \\
 & | & \texttt{fun}(x) \rightarrow \texttt{ ... end} \\
\Gamma, xs \in \mathbb{L} & \vdash & \text{skel}(\bar{s}, xs) \\
 & = & \texttt{skel : do}(\bar{s}, xs) \\
\Gamma & \vdash & \bar{f} \\
 & = & \{\texttt{func},\ \mathcal{F}\} \\
\Gamma & \vdash & pipe(\bar{s}_1, \ldots, \bar{s}_2) \\
 & = & \{\texttt{pipe}, [\bar{s}_1, \ldots, \bar{s}_n]\} \\
\Gamma & \vdash & farm(\bar{s}, n) \\
 & = & \{\texttt{farm}, \bar{s}, n\} \\
\Gamma & \vdash & map(\mathcal{F}, xs) \\
 & = & \texttt{lists : map}(\mathcal{F}, xs)
\end{array}
$$

Figure 2. Semantic Equivalences for Refactoring Notation; see Fig. 1 for environment definitions.

Refactoring	Description
Compose Maps	Lifts a series of map operations $m_1, m_2, \ldots, m_{n-1}, m_n$ such that the input of m_i is the output of m_{i-1} where $1 < i < n$, into a single anonymous function composing the functions of the map operations respectively. The function is then assigned to some user-provided variable name.
Intro Skel	Given some skeleton configuration, introduces a call to the Skel library over some map operation or list comprehension.
Intro Feedback	Transforms some map-equivalent recursive function containing a call to Skel, such that the call to Skel is updated with a feedback skeleton, removing the outer recursive call.

Figure 3. Textual Overview of Refactorings

The general case is defined:

$$
\begin{array}{rcl}
\Gamma, F \notin \Gamma, \mathcal{A}_{\mathcal{E}_x} & \vdash & R = [\mathcal{E}_x \mid\mid x \leftarrow xs] \\
 & \mapsto & F = fun(x,\ \mathcal{A}_{\mathcal{E}_x}) \rightarrow \\
 & & \qquad \mathcal{E}_x \\
 & & end, \\
 & & R = map(F, xs)
\end{array}
$$

Where F is a user-provided valid variable name; $\mathcal{E}_X$ denotes a set of Erlang statements parametrised over x; and $\mathcal{A}_{\mathcal{E}_x}$ denotes the set of non-x arguments required by $\mathcal{E}_x$.

Under the general case, the statements that form the output expression are wrapped in an anonymous function and parametrised according to the variable dependences of $\mathcal{E}_x$. This function is assigned to a user-provided variable name. Only one input variable may be provided by the input set. The result is a two-statement closure containing the lifted output expression and a map operation which applies the original input to the newly-created function.

Whilst the above is the definition of the general case, the refactoring is extended by variations accounting for different list comprehensions. One variation, for example, lifts and assigns multiple nested functions. Given the code:

```
R = [f(g(X)) || X <- Xs]
```

it is possible to lift f and g into their own functions, producing:

```
RG = fun(X) -> g(X) end,
RF = fun(X) -> f(X) end,
Q = lists:map(RG, Xs),
R = lists:map(RF, Q)
```

3.2 Introduce Func

Once atomic sequential closures have been identified, perhaps through *Extract Composition Function* (3.1), it is necessary wrap the closures in a `func` skeleton to enable their use in Skel. We again note that whilst other skeleton-introducing refactorings include this refactoring as part of their operation, the basic *Introduce Func* skeleton allows for greater manipulation once all components are ready to be arranged into the invocation of Skel.

Introduce Func is defined as follows:

$$\Gamma \quad \vdash \quad \mathcal{F} \mapsto \bar{f}$$

Where, as defined in Fig. 2, $\mathcal{F}$ denotes the multiple possible representations of a function in Erlang with an arity of 1. Under *Introduce Func*, $\mathcal{F}$ will not be transformed should it be part of any statement where its wrapping in a `func` skeleton would lead to syntactic errors.

4. Refactoring the Multi-Agent System

We demonstrate how program shaping can be used by illustrating its application to a multi-agent system use case developed by AGH. The system 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.

```
loop(Islands, Time, SP, Cf) ->
   Tag = fun(Island) ->
              [{
                mas_misc_util:behaviour_proxy(
                  Agent, SP, Cf),
                Agent} || Agent <- Island]
        end,

   Groups = [mas_misc_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_misc_util:meeting_proxy(
               Activity,
               mas_sequential,
               SP,
               Cf) || Activity <- I]
             || I <- Groups],

   WithMigrants = append(
                lists:flatten(Migrants),
                NewGroups),

   NewIslands = [mas_misc_util:shuffle(lists:flatten(I)
        )
              || I <- WithMigrants],

   case os:timestamp() < Time of
      true ->
         loop(NewIslands, Time, SP, Cf);
      false ->
         NewIslands
   end.
```

Whilst this code is a good candidate for parallelisation, it cannot be parallelised immediately. It needs to be *shaped* first. We illustrate the method by which this code may be shaped using pre-existing refactorings, and refactorings defined in Section 3.

4.1 Stage 1

We start shaping `loop/4` by extracting functions from the list comprehensions assigned to `Tagged`, `Groups`, and `Migrants` using *Extract Comprehension Function*, producing the following code.

```
loop(Islands, Time, SP, Cf) ->
   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)))),

   NewGroups = [[mas_misc_util:meeting_proxy(
                Activity,
                mas_sequential,
                SP,
                Cf) || Activity <- I]
             || I <- Groups],

   WithMigrants = append(
                 lists:flatten(Migrants),
                 NewGroups),

   NewIslands =
      [mas_misc_util:shuffle(lists:flatten(I))
       || I <- WithMigrants],

   case os:timestamp() < Time of
      true ->
         loop(NewIslands, Time, SP, Cf);
      false ->
         NewIslands
   end.
```

4.2 Stage 2

To facilitate its eventual composition with `TagFun` and `GroupFun`, we inline `MigrantsFun` using the classical `Inline Method` refactoring.

```
loop(Islands, Time, SP, Cf) ->
```

```
    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 ({{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)))),

    NewGroups = [[mas_misc_util:meeting_proxy(
                    Activity,
                    mas_sequential,
                    SP,
                    Cf) || Activity <- I]
                 || I <- Groups],

    WithMigrants = append(
                     lists:flatten(Migrants),
                     NewGroups),

    NewIslands = [mas_misc_util:shuffle(lists:flatten(I)
         )
                  || I <- WithMigrants],

    case os:timestamp() < Time of
        true ->
            loop(NewIslands, Time, SP, Cf);
        false ->
            NewIslands
    end.
```

4.3 Stage 3

As `MigrantsFun` can now be composed with `TagFun` and `GroupFun`, we compose these three functions using *Compose Functions*.

```
loop(Islands, Time, SP, Cf) ->
    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 = lists:map(TGM, Islands),

    NewGroups = [[mas_misc_util:meeting_proxy(
                    Activity,
                    mas_sequential,
                    SP,
                    Cf) || Activity <- I]
                 || I <- TGMs],

    NewIslands = [mas_misc_util:shuffle(lists:flatten(I)
         )
                  || I <- NewGroups],

    case os:timestamp() < Time of
        true ->
            loop(NewIslands, Time, SP, Cf);
        false ->
            NewIslands
    end.
```

Here we note that the input to the list comprehension assigned to `NewGroups` has been changed according to the newly introduced composition. Similarly we also remove `WithMigrants` with the `Remove Statement` refactoring, which requires the input to the list comprehension assigned to `NewIslands` to be changed to `NewGroups`.

4.4 Stage 4

We next focus our attention on the list comprehensions assigned to `NewGroups` and `NewIslands` respectively, applying *Extract Comprehension Function* to both.

```
loop(Islands, Time, SP, Cf) ->
    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 = lists:map(TGM, Islands),

    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),

    case os:timestamp() < Time of
        true ->
            loop(NewIslands, Time, SP, Cf);
        false ->
            NewIslands
    end.
```

4.5 Stage 5

Having shaped our existing functions into a suitable form for parallelisation, we now proceed to introduce the structures to pass to Skel. We start with the map operation which applies `TGM` to each element in `Islands`, transforming it using *Intro Func* to introduce a `func` skeleton. We apply the same refactoring to the `NewGroupsInnerFun` expression. Continuing this process, we next apply *Intro Farm* over the `NewGroupsFun` expression.

```
loop(Islands, Time, SP, Cf) ->
    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},
    NewGroups = lists:map(NewGroupsFun, TGMs),

    Shuffle =
        fun (I) ->
                mas_misc_util:shuffle(lists:flatten(I))
        end,
    NewIslands = lists:map(Shuffle, NewGroups),

    case os:timestamp() < Time of
        true ->
            loop(NewIslands, Time, SP, Cf);
        false ->
            NewIslands
    end.
```

To aid readability we also rename `NewIslandsFun` to `Shuffle`.

4.6 Stage 6

We again apply *Intro Seq*, this time over the renamed `Shuffle` expression, completing all skeletons required to introduce the invocation to Skel. As such, we apply *Intro Skel* over `NewIslands` and `NewGroups`.

```
loop(Islands, Time, SP, Cf) ->
    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]},
    NewIslands =
        [NewIsland ||
            {_, NewIsland} <- skel:do([Pipe], Islands)],

    case os:timestamp() < Time of
        true ->
            loop(NewIslands, Time, SP, Cf);
        false ->
            NewIslands
    end.
```

4.7 Stage 7

Whilst `loop/4` is now parallel, the outer loop itself can be folded into the Skel invocation for efficiency. We do this by applying *Intro Feedback Loop* over `loop/4` itself.

```
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]).
```

This completes the shaping and parallelisation process.

5. Results

5.1 Performance Improvements

The plots in Figure 4 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 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.

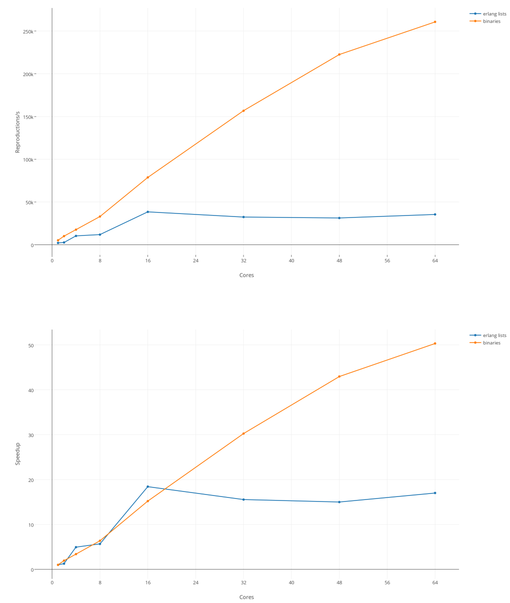

Figure 4. Performance Results for Binary and List for Multi-Agent System

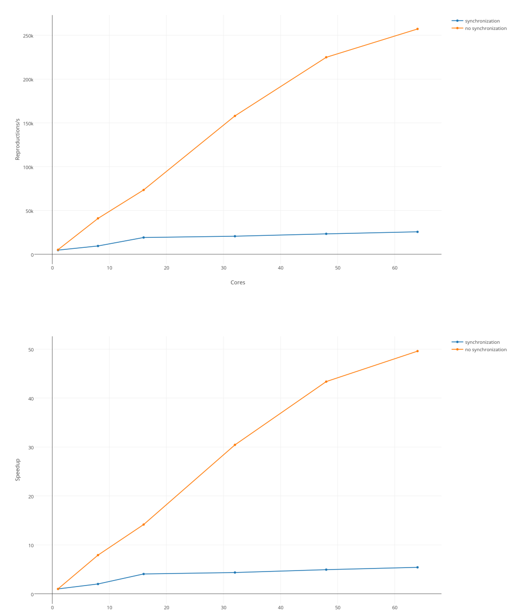

Figure 5. Performance Results for Further Tuned Multi-Agent System

6. Related Work

The study of parallelism has a long and active history; often demonstrating the difficulties associated with the style, and illustrating its core requirements [30]. Approaches designed to simplify its introduction and management are numerous and varied; examples include: futures [12], strategies [33], monads [24], and algorithmic skeletons [8]. Each approach is similar such that low-level parallel mechanics are hidden from the programmer, and that each have requirements for their introduction. These requirements are unlikely to be met without the need for program transformation.

As with parallelism, the study of program transformation is not a new area, with Partsch and Steinbrüggen describing early work in 1983 [29], and more recently by Mens in 2004 [27]. Not limited to the imperative style, refactoring tools have been built for both Haskell and Erlang [17, 19].

Despite the work done for both algorithmic skeletons and program transformation, there have only been limited attempts at combining the two [16]. Some attempts include high-level pattern-based rewrites including extensions to Haskell's refactoring tool HaRe [3], and similar, cost-directed refactorings for Wrangler [4]. These extensions are limited by the number of refactorings they include, and their focus on the introduction and manipulation of skeleton library invocations. Transformations that allow the introduction of high-level parallel libraries remain a predominantly manual process.

7. Conclusions and Future Work

Whilst many continue to write sequential software, the shift to parallel hardware requires the modern programmer to *think parallel*. Although advances in structured parallel techniques greatly simplify the task of introducing the mechanics of parallelism, these techniques do not immediately fit every program.

We have presented how refactoring and program shaping techniques can be employed alongside the Skel library, an Erlang implementation of several algorithmic skeletons, to restructure and introduce parallel to an Erlang evolutionary multi-agent system, a real world use case. Following this transformation, we have evaluated the effectiveness of the resulting program in terms of performance gains, discovering speedups of 19–50% depending on representation of agents.

In future work, we intend on applying this technique to other use cases and further evaluating its effectiveness; the Erlang Dialyzer, for example. It is also our intention to expand our library of program shaping techniques, incorporating static analysis techniques to further automate the process, at the same time reducing the burden on the programmer.

References

[1] I. Bozó, V. Fordós, Z. Horvath, M. Tóth, D. Horpácsi, T. Kozsik, J. Köszegi, A. Barwell, C. Brown, and K. Hammond. Discovering Parallel Pattern Candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 13–23, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3038-1. . URL `http://doi.acm.org/10.1145/2633448.2633453`.

[2] C. Brown and S. Thompson. Clone Detection and Elimination for Haskell. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 111–120, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-727-1.

[3] C. Brown, H. Li, and S. Thompson. An Expression Processor: A Case Study in Refactoring Haskell Programs. In R. Page, editor, *Eleventh Symposium on Trends in Functional Programming*, page 15pp, May 2010.

[4] C. Brown, M. Danelutto, K. Hammond, P. Kilpatrick, and A. Elliott. Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming*, pages 1–19, 2013. ISSN 0885-7458. . URL `http://dx.doi.org/10.1007/s10766-013-0266-5`.

[5] C. Brown, V. Janjic, M. Goli, K. Hammond, and J. McCall. Bridging the Divide: Intelligent Mapping for the Heterogeneous Programmer. In *High-Level Programming for Heterogeneous and Hierarchical Parallel Systems*, 2013.

[6] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J. ACM*, 24(1):44–67, 1977.

[7] E. Cantú-Paz. A Survey of Parallel Genetic Algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.

[8] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, Mar. 2004. ISSN 0167-8191.

[9] M. I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD thesis, 1988. AAID-85022.

[10] D. Dig. A Refactoring Approach to Parallelism. *Software, IEEE*, 28 (1):17–22, Jan 2011. ISSN 0740-7459. .

[11] D. Fields, S. Saunders, and E. Belyaev. *IntelliJ IDEA in Action*. In Action. Manning, 2006. ISBN 9781932394443.

[12] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A Heterogeneous Parallel Language. In *Proc. of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 37–44, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[14] J. George, M. Gleizes, P.Glize, and C.Regis. Real-time simulation for flood forecast: an adaptive multi-agent system staff. In *Proceedings of the AISB'03 Symposium on Adaptive Agents and Multi-Agent Systems*. University of Wales, 2003.

[15] González-Vélez, Horacio and Leyton, Mario. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, Nov. 2010. ISSN 0038-0644.

[16] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. González-Vélez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. In *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 218–236. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35886-9. .

[17] T. Kozsik, Z. Csrnyei, Z. Horvth, R. Kirly, R. Kitlei, L. Lvei, T. Nagy, M. Tth, and A. Vg. Use Cases for Refactoring in Erlang. In *Central European Functional Programming School*, volume 5161 of *Lecture Notes in Computer Science*, pages 250–285. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-88058-5.

[18] D. Krzywicki, W. Turek, A. Byrski, and M. Kisiel-Dorohinicki. Massively-concurrent Agent-based Evolutionary Computing. *CoRR*, abs/1501.06721, 2015.

[19] H. Li and S. Thompson. Tool Support for Refactoring Functional Programs. In *PEPM '08*, pages 199–203, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-977-7.

[20] H. Li and S. Thompson. A domain-specific language for scripting refactorings in erlang. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 501–515, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28871-5.

[21] H. Li, S. J. Thompson, and C. Reinke. The Haskell Refactorer, HaRe, and its API. *Electr. Notes Theor. Comput. Sci.*, 141(4):29–34, 2005.

[22] H. Li, S. Thompson, G. Orosz, and M. Tóth. Refactoring with Wrangler, Updated: Data and Process Refactorings, and Integration with Eclipse. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 61–72, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4.

[23] S. Marlow. *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. " O'Reilly Media, Inc.", 2013.

[24] S. Marlow, R. Newton, and S. Peyton Jones. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1. . URL `http://doi.acm.org/10.1145/2034675.2034685`.

[25] S. McArthur, V. Catterson, and N. Hatziargyriou. Multi-agent systems for power engineering applications part i: Concepts, approaches, and technical challenges. *IEEE TRANSACTIONS ON POWER SYSTEMS*, 22(4), November 2007.

[26] M. McCool, A. Robison, and J. Reinders. *Structured Parallel Programming*. Morgan Kaufmann, 2012. ISBN 978-0-124-15993-8.

[27] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, Feb. 2004. ISSN 0098-5589. . URL `http://dx.doi.org/10.1109/TSE.2004.1265817`.

[28] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, 1992.

[29] H. Partsch and R. Steinbrüggen. Program Transformation Systems. *ACM Comput. Surv.*, 15(3):199–236, Sept. 1983. ISSN 0360-0300.

[30] D. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, New York, NY, USA, 1995. ISBN 0-521-45511-1.

[31] N. Sultana and S. Thompson. Mechanical Verification of Refactorings. In *Workshop on Partial Evaluation and Program Manipulation*, pages 182–196. ACM SIGPLAN, January 2008. URL `http://www.cs.kent.ac.uk/pubs/2008/2635`.

[32] P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: A Portable Parallel Implementation of Haskell. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 79–88, New York, NY, USA, 1996. ACM. ISBN 0-89791-795-2. . URL `http://doi.acm.org/10.1145/231379.231392`.

[33] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Funct. Program.*, 8(1):23–60, Jan. 1998. ISSN 0956-7968. . URL `http://dx.doi.org/10.1017/S0956796897002967`.

[34] P. Uhruski, M. Grochowski, and R. Schaefer. A Two-layer Agent-Based System For Large-Scale Distributed Computation. *Computational Intelligence*, 24(3):191–212, July 2008.

[35] M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2009.