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...
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, \( f : a \rightarrow b \), 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 independently 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 problems 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 varied, the approach and design of under-
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
techniques 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 \( \gamma \) allowing
access and reference to the current scope of the rewrite rule within
the source program. This includes the set of all available functions
\( F \). The skeleton library Skel, and the skeletons it provides are
denoted by the set \( S \).

\[
S = \{ \text{skel, } f, \text{pipe, } farm, \text{farm'}, \text{farm'\text{'}, cluster, cluster'}, \text{cluster''} \}
\]

For all rewrite rules, \( S \) is assumed to be in scope. This denoted in
each rule by extending \( \gamma \):

\[
\gamma = \gamma \cup S
\]

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:

\[
\bar{s}, xs \in \Gamma, xs: \text{list} \vdash \text{skel}(\bar{s}, xs) = \text{skel}: \text{do}(\bar{s}, xs)
\]

Where \( \bar{s} \) represents any valid skeleton in \( S \), i.e. \( S \setminus \{ \text{skel} \} \); 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.

\[
R = [f(g(X)) | X \leftarrow Xs]
\]
it is possible to lift \( f \) and \( g \) into their own functions, producing:

\[
\begin{align*}
\text{RG} & = \text{fun}(X) \rightarrow g(X) \end{align*}
\]

\[
\begin{align*}
\text{RF} & = \text{fun}(X) \rightarrow f(X) \end{align*}
\]

\[
\begin{align*}
\text{Q} & = \text{lists:map(RG, Xs)}, \\
\text{R} & = \text{lists:map(RF, Q)}
\end{align*}
\]

3.2 Introduce Func

Once atomic sequential closures have been identified, perhaps through \textit{Extract Composition Function} (3.1), it is necessary wrap the closures in a \textit{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 \textit{Introduce Func} skeleton allows for greater manipulation once all components are ready to be arranged into the invocation of Skel.

\[ \Gamma \vdash F \mapsto \bar{f} \]

Where, as defined in Fig. 2, \( \bar{F} \) denotes the multiple possible representations of a function in Erlang with an arity of 1. Under \textit{Introduce Func}, \( \bar{F} \) will not be transformed should it be part of any statement where its wrapping in a \textit{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.

\[
\begin{align*}
\text{loop}(\text{Islands}, \text{Time}, \text{SP}, \text{Cf}) & \rightarrow \\
\text{Tag} & = \text{fun} (\text{Island}) \rightarrow \\
& \begin{cases} \\
\{\text{mas_misc_util:behaviour_proxy(Agent, SP, Cf), Agent} \mid \text{Agent} \leftarrow \text{Island}\} \\
\end{cases} \\
\text{Groups} & = \text{mas_misc_util:group_by(Tag(I))} \mid I \leftarrow \text{Islands}, \\
\text{Migrants} & = \text{seq_migrate(lists:keyfind(migration, 1, Island), Nr)} \mid \{\text{Island, Nr} \leftarrow \\
& \begin{cases} \\
\text{lists:zip} (\text{Groups}, \\
\text{lists:seq(1, length(Groups))))}, \\
\end{cases} \\
\text{NewGroups} & = \text{mas_misc_util:meeting_proxy(Activity, mas_sequential, SP, Cf)} \mid \text{Activity} \leftarrow I \mid I \leftarrow \text{Groups}, \\
\text{WithMigrants} & = \text{append} (\text{lists:flatten(Migrants), NewGroups}), \\
\text{NewIslands} & = \text{mas_misc_util:shuffle(lists:flatten(I))} \mid I \leftarrow \text{WithMigrants}, \\
\end{align*}
\]

Whilst this code is a good candidate for parallelisation, it cannot be parallelised immediately. It needs to be \textit{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 \( \text{loop/4} \) by extracting functions from the list comprehensions assigned to \textit{Tagged}, \textit{Groups}, and \textit{Migrants} using \textit{Extract Comprehension Function}, producing the following code.

\[
\begin{align*}
\text{loop}(\text{Islands, Time, SP, Cf}) & \rightarrow \\
\text{TagFun} & = \text{fun} (\text{Agent}) \rightarrow \\
& \{\text{mas_misc_util:behaviour_proxy(Agent, SP, Cf)}, \text{Agent}\}, \\
\text{Tagged} & = \text{lists:map(TagFun, Islands)}, \\
\text{GroupFun} & = \text{fun} (I) \rightarrow \text{mas_misc_util:group_by(I)} \end{align*}
\]

\[
\begin{align*}
\text{Groups} & = \text{lists:map(GroupFun, Tagged)}, \\
\text{MigrantFun} & = \text{fun} \{\text{Island, Nr}\} \rightarrow \\
& \text{seq_migrate(lists:keyfind(migration, 1, Island), Nr)}, \\
\text{Migrants} & = \text{lists:map(MigrantFun, lists:zip(Groups, lists:seq(1, length(Groups)))))}, \\
\text{NewGroups} & = \text{mas_misc_util:meeting_proxy(Activity, mas_sequential, SP, Cf)} \mid \text{Activity} \leftarrow I \mid I \leftarrow \text{Groups}, \\
\text{NewIslands} & = \text{mas_misc_util:shuffle(lists:flatten(I))} \mid I \leftarrow \text{WithMigrants}, \\
\end{align*}
\]

4.2 Stage 2

To facilitate its eventual composition with \textit{TagFun} and \textit{GroupFun}, we inline \textit{MigrantsFun} using the classical \textit{Inline Method} refactoring.
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 <- 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.

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,
    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 <- 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.

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,
    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))))
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,
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.

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.

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.
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.
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 Diary, 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


