Computing and Informatics, Vol. 35, 2016, 792–818

USING PROGRAM SHAPING AND ALGORITHMIC SKELETONS TO PARALLELISE AN EVOLUTIONARY MULTI-AGENT SYSTEM IN ERLANG

Adam D. Barwell, Christopher Brown, Kevin Hammond

School of Computer Science
The University of St Andrews
St Andrews, U.K.
e-mail: {adb23, cmb21, kh8}@st-andrews.ac.uk

Wojciech Turek, Aleksander Byrski

Department of Computer Science
Faculty of Computer Science, Electronics and Telecommunications
AGH University of Science and Technology, Kraków, Poland
e-mail: {wojciech.turek, olekb}@agh.edu.pl

Abstract. This paper considers how to use program shaping and algorithmic skeletons to parallelise a multi-agent system that is written in Erlang. Program shaping is the process of transforming a program to better enable the introduction of parallelism. Whilst algorithmic skeletons abstract away the low-level aspects of parallel programming that often plague traditional techniques, it is not always easy to introduce them into an arbitrary program, especially one that has not been written with parallelism in mind. Amongst other issues, data may not always be in a compatible format, function calls may need to be replicated to support alternative uses, side-effects may need to be isolated, or there may be dependencies between functions and data that obstruct the introduction of parallelism. Program shaping can be used to transform such code to a form that allows skeletons to be more easily introduced. In this paper, we present a series of generic program shaping rewrite rules, provide their implementation as refactorings, and show how they can be used to parallelise an Evolutionary Multi-Agent System (MAS) written in Erlang. We show that we can significantly speed up this application, obtaining super-linear speedups of over $70\times$ the original sequential performance on a 64-core shared-memory machine.

1 INTRODUCTION

Parallel processors are now ubiquitous. Consequently, parallel programming is an increasingly 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 detailed parallelism constructs, e.g. threads, communication, locking, and synchronisation. This results in a process which is often tedious, difficult, and error-prone [27]. In response, a number of approaches have been devised that aim to simplify the parallelisation process (e.g. [17, 39, 40, 41, 49]). While these works take different approaches, they all abstract away low-level parallel mechanics, which is a common source of error. Although beneficial, these abstractions only serve to address one specific problem: that the interfaces to parallel primitives and libraries are often too low-level. Other aspects of parallelism, such as which part(s) of a program should be parallelised [1] or which parallel configuration gives the best performance gains [31], must be similarly addressed. One common and non-trivial aspect of parallelisation, which has so far received little attention, is how to restructure code the best to enable the introduction of parallelism. This may involve e.g. the detection and breaking of inter-task dependencies, the detection of side-effects that inhibit parallelism, changing data representations to avoid excessive memory use, and/or avoiding scheduler inefficiencies. This requires extensive knowledge of the program code, the language used, and parallelism itself. The difficulty of this task makes it a significant stage of the parallelisation process. This restructuring stage is a form of *program shaping*, which we define to be *a series of intentional changes to source code that contribute towards some goal.*

At present, program shaping is generally a manual, *ad hoc*, and error-prone process. This paper investigates how to increase the automation of the program shaping task, using *refactoring*. A *refactoring* [23] is a conditional, source-to-source program transformation that maintains functional correctness. While manual refactorings are possible, refactoring tools can perform transformations both automatically and *safely*. Such tools exist for a wide range of languages, and are often integrated with popular editors [42]. In the functional programming community, well-known examples include Wrangler [37] and RefactorErl [28] for Erlang, and HaRe for Haskell [38]. A recent work has demonstrated that refactoring can be used to automate the introduction of parallelism [2]. Where that work focuses on introducing code to invoke parallel operations, this paper extends the idea to program shaping, presenting refactorings that restructure programs to better facilitate the introduction of parallel constructs.

While fully automatic approaches exist, and are generally desirable, they are often limited by the patterns they can operate on. Conversely, algorithmic skeletons and similar abstraction approaches are very flexible but depend upon the programmer to introduce them. Program shaping acts as an effective bridge between these two approaches; offering increased automation to reduce the programmer's burden while maintaining the flexibility of skeletons through a programmer guidance of their

application. We demonstrate how program shaping can be applied to a large-scale evolutionary multi-agent system (EMAS) written in Erlang.

"Evolution" means that agents are able to *reproduce* (generate new agents), which is a kind of cooperative interaction, and may *die* (be eliminated from the system), which is the result of competition (selection). The idea of an Evolutionary Multi-Agent System was first introduced by Cetnarowicz in 1996 [15]. Since then it has been implemented a number of times (e.g. [8, 20]), analysed [10, 9, 45], and extended [46, 7, 13]. Evolutionary Multi-Agent Systems have turned out to be an efficient paradigm for solving a variety of complex optimisation problems [52, 11].

Evolutionary processes are by nature decentralized and therefore evolutionary processes may be easily introduced in a multi-agent system at a population level. In this paper, we demonstrate how such an evolutionary MAS, built in Erlang, might be semi-automatically restructured and parallelised using new automated refactoring techniques for program shaping. We provide a description of the refactorings used below, and demonstrate that, in combination with the use of algorithmic skeletons, we can achieve excellent performance gains (in the best case, over 70 ×, on a 64-core multicore system).

2 BACKGROUND

2.1 Algorithmic Skeletons

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction [17] 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, where the 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. A recent survey of algorithmic skeleton approaches can be found in [25]. 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 through 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 alternative hardware architectures (especially where an application targets different classes of architecture).

2.1.1 Skel

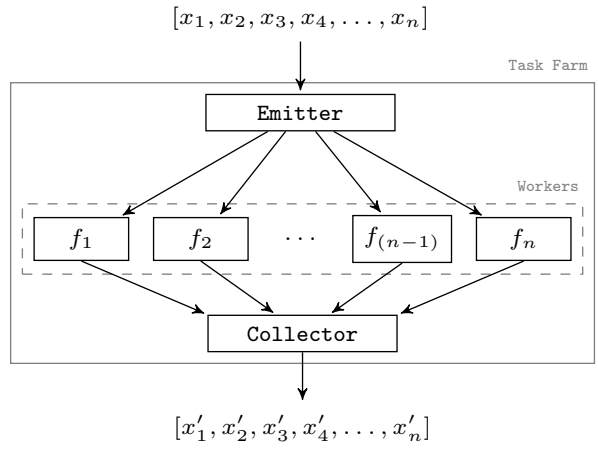


Figure 1. A Task Farm Skeleton

The Skel [2] 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 (see Figure 1) 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

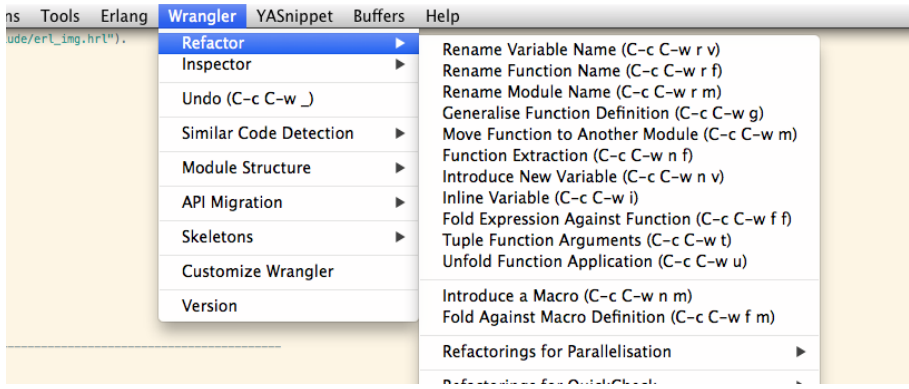


Figure 2. Wrangler's menu in Emacs

program functionality, and it is generally applied semi-automatically, i.e. under programmer direction. This allows programmer knowledge about e.g. safety properties to be exploited, and so permits a wider range of possible transformation than a fully automatic approach. Refactorings can be described as either elementary or composite. An elementary refactoring is a single transformation that maintains functional correctness. A composite refactoring may be composed of one or more elementary or composite refactorings, and where intermediate transformations may not preserve functional correctness, the composite refactoring taken as a whole does that. We assume that the functions under refactoring are pure and do not contain side effects, checking for these side effects as part of the condition checking is currently future work. 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 [5].
- 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 [48].

The term "refactoring" was first introduced by Opdyke in his 1992 PhD thesis [43], 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 the IntelliJ IDEA refactorer supports 35 distinct refactorings for Java [21]. Typical refactorings include *variable renaming* (changing all instances of a variable that are in scope to a new name), *parameter adding* (introducing a new parameter to a function definition and updating all relevant calls to that function with a placeholder), *function extraction* (lifting a selected block of code into its own function) and *function inlining* (replacing the invocation of a function with the body of that function).

2.2.1 Wrangler

While the refactoring community has produced a great deal of work for the object-oriented paradigm [18], the concept can be applied to a wide range of programming styles and approaches, including functional programming. Indeed, Darlington and Burstall's transformation system for recursive functions produces code that would not be out of place in modern functional programs [6]. Li et al. have since produced the HaRe [38] and Wrangler [37] 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 (Figure 2) and Eclipse. In this paper, 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 [35]; plus a Domain-Specific Language (DSL) to compose the refactorings [35].

2.3 Evolutionary Multi-Agent System

A common approach to problem solving is to decompose that problem into smaller tasks, to solve each (sub)task independently, and to subsequently combine the 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 [14]. 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 [51]. One of the major application areas of multi-agent systems is large-

scale computing [50]. Evolutionary multi-agent systems are a hybrid meta-heuristic which combines multi-agent systems with evolutionary algorithms. The idea consists of evolving a population of agents to improve their ability to solve a particular optimisation problem [12, 10].

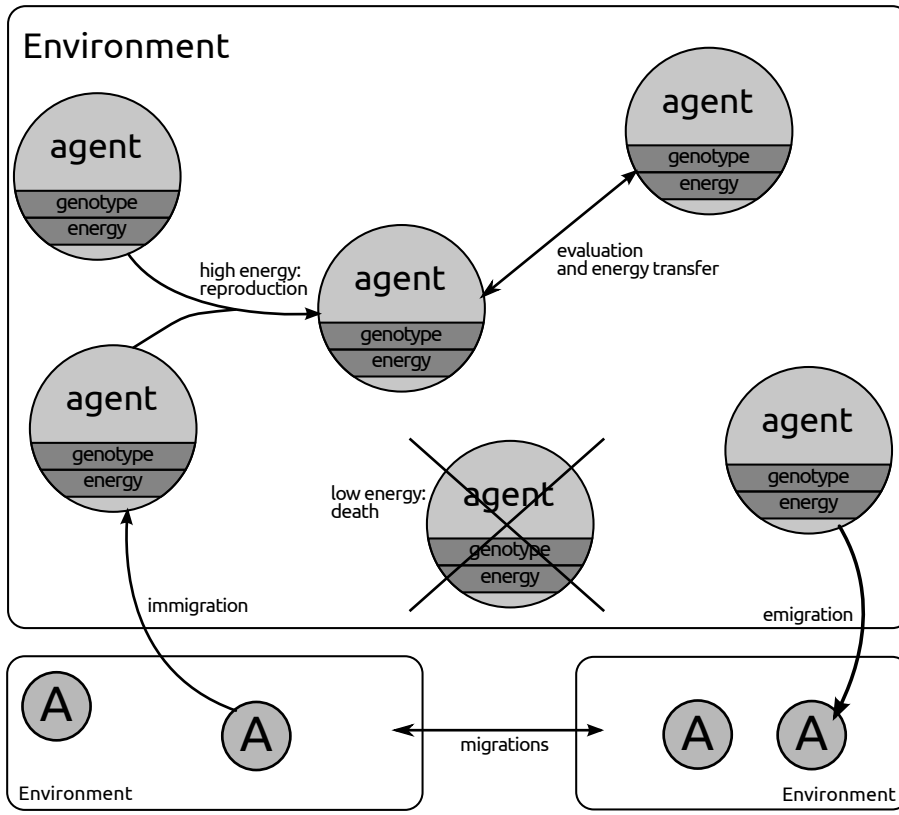


Figure 3. EMAS structure and principle of work

In a multi-agent system, no global knowledge is available to individual agents. Agents should remain autonomous and no central authority should be needed. Therefore, in an evolutionary computing system, unlike traditional evolutionary algorithms, selective pressure needs to be decentralised. Using agent terminology, we can say that selective pressure is required to emerge from peer-to-peer interactions between agents instead of being globally-driven. In a basic algorithm, every agent is assigned a real-valued vector that represents a potential solution to the optimisation problem, together with the corresponding fitness measure. Emergent selective pressure is achieved by giving agents a single non-renewable resource called energy. Agents start with an initial amount of energy and meet randomly. If their energy is below a threshold, they fight by comparing their fitness – better agents

take energy from worse ones. Otherwise, the agents reproduce and yield a new one – the genotype of the child is derived from its parents using variation operators and it also receives some energy from its parents. Since the total energy remains constant, the system is stable. However, the number of agents may vary and adapt to the difficulty of the problem (see Figure 3). As in other evolutionary algorithms, agents can be split into separate populations. Such sub-populations, called islands, help preserve diversity by introducing allopatric speciation and can also execute in parallel. Information is exchanged between islands through agent migrations. It should be noted that the EMAS computing abilities have been formally proven correct by constructing a detailed Markov-chain based model and proving its ergodicity property [10]. This result shows that EMAS is a general optimisation tool.

While the problems that can be solved by both evolutionary and standard multi-agent systems are varied, the approach and design of the underlying systems is often standard. The recent work by Krzywicki et al. [33] has developed patterns to describe the operation of evolutionary and multi-agent systems, with an example implementation in Erlang. While this implementation was initially completely sequential, through the use of the program shaping techniques and methodology described below, we have now been able to parallelise it successfully to give an efficient and scalable parallel implementation.

3 REWRITE RULES

In this section, we define a series of program shaping refactorings. All the 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 that ensure that the transformation is valid, a description of the syntax to be transformed, and a description of the revised syntax following the 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 $\mathbb{S}$.

$$\mathbb{S} = \{skel, \bar{f}, pipe, farm\}.$$

For all rewrite rules, $\mathbb{S}$ is assumed to be in scope. This is 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, given in Figure 4. They are in the form:

$$\Gamma \vdash \bar{f} = \{\texttt{func},\ \mathcal{F}\}$$

where Γ represents the environment under which the rule is valid, $\bar{f}$ is the simplified expression that represents, and can be rewritten as, $\{\texttt{func},\ \mathcal{F}\}$. 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 (*Extract Composition Function* and *Introduce Func*), giving textual descriptions of the others in Figure 5.

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

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

Figure 4. Environment definitions and semantic equivalences for refactoring notation

3.1 Extract Composition Function

The *Extract Composition Function* refactoring exposes sequential functionality that may later be used as part of a parallel pipeline. While it is possible to immediately introduce a skeleton over a list comprehension, e.g. via *Intro Farm*, such refactorings commonly assume that the list comprehension is the top-level command. Where the

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 5. Textual overview of refactorings

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.

The general case is defined as shown below:

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

where F is a valid user-provided variable name; $\mathcal{E}_X$ denotes a set of Erlang statements parameterised over x; and $\mathcal{A}_{\mathcal{E}_x}$ denotes the set of non-x arguments required by $\mathcal{E}_x$. In the general case, the statements that form the output expression are wrapped in an anonymous function and parameterised according to the variable dependencies 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. The refactoring can be extended to special cases that account for e.g. specific forms of list comprehension. For example, one variant lifts and assigns multiple nested functions. Given the code:

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

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

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

3.2 Introduce Func

Once atomic sequential closures have been identified, perhaps through *Extract Composition Function*, it is then necessary to wrap the closures in a `func` skeleton in order to enable their use in Skel. *Introduce Func* is defined as follows:

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

where, as defined in Figure 4, $\mathcal{F}$ denotes the multiple possible representations of an Erlang function whose arity is 1. $\mathcal{F}$ will not be transformed if wrapping it into 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 an example MAS. The MAS operates over a number of generations to find a solution. Each generation may be modelled as an iteration of a loop, with each member of the system's population performing its own work within each iteration. Both the outer generational loop and the work performed within that loop are highly suitable for parallelisation. The code below has been slightly simplified for readability.

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

While this code seems to be a good candidate for parallelisation, it cannot be parallelised immediately, and so needs to be *shaped* first. We show how this code may be shaped using some standard refactorings, plus the new refactorings from Section 3. We give only the affected code for each stage of the transformation.

4.1 Stage 1

We start shaping `loop/4` by extracting functions from the list comprehensions that are assigned to `Tagged`, `Groups`, and `Migrants` using the *Extract Comprehension Function* refactoring. This gives the following definitions.

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

4.2 Stage 2

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

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

4.3 Stage 3

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

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

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 the *Extract Comprehension Function* refactoring to both.

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

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 the *Intro Func* refactoring to introduce a `func` skeleton. We apply the same refactoring to the `NewGroupsInnerFun` expression. Continuing this process, we next apply the *Intro Farm* refactoring over the `NewGroupsFun` expression.

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

In order to aid readability, we also rename `NewIslandsFun` to `Shuffle`.

4.6 Stage 6

We again apply the *Intro Func* refactoring, this time over the renamed `Shuffle` expression, completing all the skeletons that are required to introduce the Skel invocation. We can then apply the *Intro Skel* refactoring over `NewIslands` and `NewGroups`.

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

4.7 Stage 7

While `loop/4` is now parallel, the outer loop itself can also be folded into the Skel invocation to improve efficiency. We do this by applying the *Intro Feedback Loop* refactoring 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 = fun(Agents) ->
              Tagged = lists:map(TagFun, Agents),
              Migrants = lists:map(MigrantFun, Tagged),
              GroupFun(Migrants)
    end,
    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 PERFORMANCE EVALUATION

The aim of the parallellization process presented in this paper is to convert a given sequential code into a version efficiently using a multi-core architecture. In order to evaluate the efficiency of the refactored code we compared it to different implementations of the same algorithm, which have been created manually – without the SKEL library providing parallel patterns and without the presented program shaping methods.

Two different implementations of the Evolutionary Multi-Agent System have been created: concurrent and hybrid. The concurrent version follows Erlang good practice for writing concurrent code, which assumes creating many fine-grained processes for all individual tasks in the system. Every agent is represented by a different process and all communication uses message-passing. Agent interactions are mediated by special processes, called "meeting arenas". This version is not influenced by the target architecture of hardware – hundreds of truly concurrent agents are created each second making Erlang scheduler responsible for managing hardware. It can successfully run on one core as well as on hundreds of cores.

The hybrid version has been designed and manually tuned for best possible performance of the EMAS algorithm. The number of individual evolutionary islands is equal to the number of cores. Each evolutionary island is internally computed sequentially, while different islands use different processes. This approach limits context switching and communication to the minimum required by incidental operations of agents migrations between islands and result collecting.

Two different optimization benchmark problems have been compared: continuous and discrete. The continuous problem was the Rastrigin function [19], a common continuous benchmarking function used to compare evolutionary algorithms. This function is highly multimodal with many local minima and one global minimum equal 0 at $\vec{x} = 0$. We used a problem size (the dimension of the function) equal to 100, in a domain equal to the hypercube $[50, 50]^{100}$

As this is a continuous optimization problem, real-valued encoding was used, with Cauchy mutations and continuous recombination as genetic operators. The Rastrigin function has a simple formulation and is easy to compute. Therefore, fitness function computation is relatively cheep and the computation to communication ratio is low.

The discrete problem was Low Autocorrelation Binary Sequences (LABS) [24]. LABS is an NP-hard combinatorial problem with a very simple formulation and with many applications in telecommunication (synchronization, pulse compression, satellite and space applications, digital signal processing, high-precision interplanetary

radar measurements), meteorology (calibration of surface profile meteorology tools), physics (Ising spin glasses, configuration state analysis, statistical mechanics) and chemistry. The LABS problem has a very difficult search space and therefore fitness function computation is far more complex than in case of the Rastrigin function, making the computation to communication ratio much higher.

We ran our simulations on the ZEUS supercomputer provided by the Pl-Grid[1] infrastructure at the ACC Cyfronet AGH[2]. We used nodes with 4 AMD Opteron 6 276 processors, with up to 64 cores and 1 GB of memory. On this architecture we executed each of the three versions of the EMAS algorithm on configurations with 1, 4, 8, 16, 32, 48 and 64 cores. Each run took approximately five minutes, during which we recorded overall number of reproductions per second. Each experiment has been repeated 10 times – the charts below present average value of all runs.

Results of the system scalability during the Rastrigin function optimization are presented in Figure 6.

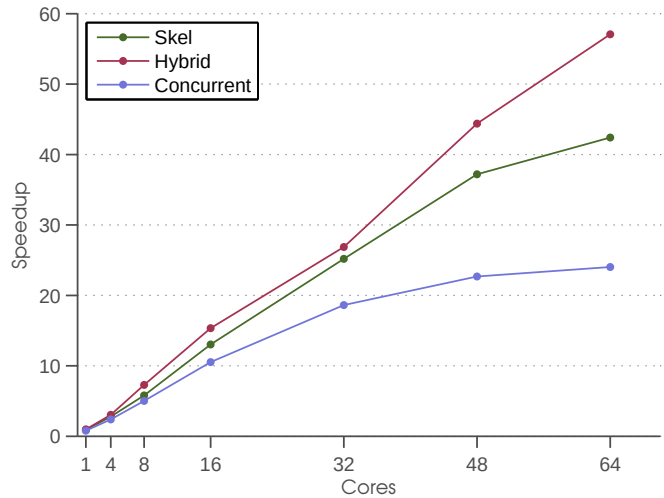


Figure 6. Speedup of the three versions of the agent-based evolutionary algorithm executing the Rastrigin function optimization

The problem with the low computation to communication ratio clearly shows that the overhead of creating hundreds of processes and passing hundreds of messages every second can reduce the scalability of the algorithm. The concurrent version scales linearly up to 16 cores only, while both hybrid and SKEL-based implementations scale linearly up to 48 cores. Hybrid version shows best characteristics in this case, outperforming the SKEL-based version especially when 64 cores are used.

[1] http://plgrid.pl/

[2] http://www.cyfronet.krakow.pl/

Results of the system scalability during the LABS problem optimization are shown in Figure 7.

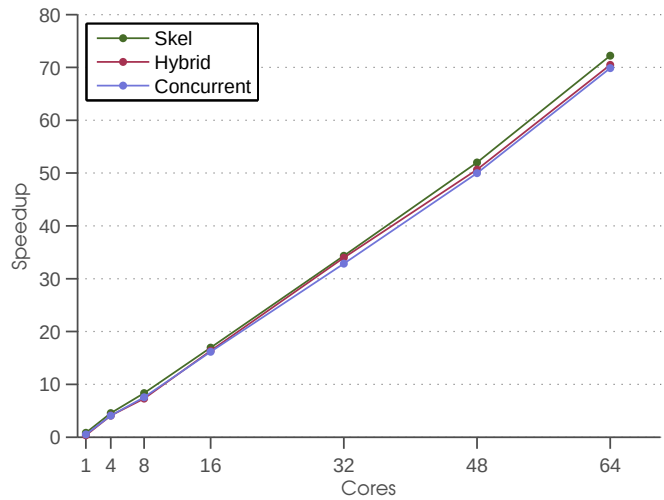


Figure 7. Speedup of the three versions of the agent-based evolutionary algorithm executing LABS problem optimization

The results clearly show that in case of the problem with high computation to communication ratio all three versions can scale linearly (or even super-linearly). The SKEL-based version showing slight superiority above the other two implementations for large numbers of cores.

The experiments confirm that the pattern-based approach for sequential code parallelization is valid and can provide efficient solutions. Highly tuned and dedicated solutions can give better results for particular problems, however as general-purpose tool the presented approach gives very good results.

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 [47]. Approaches designed to simplify its introduction and management are numerous and varied; examples include: futures [22], evaluation strategies [49], monads [40], and algorithmic skeletons [16]. Some approaches, such as futures and monads, can be language-specific, or require significant changes to the language itself. Skeletons and evaluation strategies are more generic; being reusable patterns that can be language-agnostic, and implementable using existing language components. However, evaluation strategies require the programmer to interact with those components at some point. Skeletons, conversely, hide their implementation, instead presenting a high-

level interface to the programmer, so making them more desirable here. Despite the differences between these and other approaches, each is similar in that low-level parallel mechanics are hidden to some degree from the programmer, and that each have requirements for their introduction. These requirements are unlikely to be met without the need for program transformation, however.

As with parallelism, the study of program transformation is not a new area, with Partsch and Steinbrüggen describing early work in 1983 [44], and more recently by Mens in 2004 [42]. In the functional programming community, refactoring tools have been built for both Haskell and Erlang [34, 32].

Despite the work done for both algorithmic skeletons and program transformation, there have only been limited attempts at combining the two [27]. Some attempts include high-level pattern-based rewrites including extensions to Haskell's refactoring tool HaRe [4], and similar, cost-directed refactorings for Wrangler [2]. These extensions are limited by the number of refactorings they include, and by 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.

In [2], we introduced a parallel refactoring methodology for programming Erlang programs, to facilitate the introduction of parallel skeletons. In [3] and [30], we presented parallel refactoring techniques for C++ programs. This paper goes beyond that work in introducing novel program shaping rather than pure skeleton introduction. As a technique, program shaping is relatively new and untouched by both skeletons and refactoring communities. While [29] suggests the use of "canonical forms", in which skeletons can easily be introduced, and to which equivalent code can be transformed, this work is limited by the number of forms identified. Similarly, [36] uses an analysis technique called program slicing to better inform refactorings to aid the introduction of programs. This work does not use skeletons, however, instead relying upon Erlang's concurrency primitives, restricting it to Erlang. In contrast to our work, most current research focuses in parallel refactoring on simple compile-time optimisations instead of source-to-source refactorings [26]. The approach presented in our paper therefore not only improves upon current methodologies by enabling their use on heterogeneous architectures, but also helps to introduce some automation to the previously-manual program shaping stage.

7 CONCLUSIONS AND FUTURE WORK

Although advances in structured parallel techniques greatly simplify the task of introducing the mechanics of parallelism, these techniques do not immediately fit every program. In this paper, we have introduced novel program shaping techniques and shown how they can be employed alongside the Skel library, an Erlang implementation of several algorithmic skeletons, to restructure and introduce parallelism to an Erlang implementation of an Evolutionary Multi-Agent System, a real world use case (universal optimisation meta-heuristics). However, although this technique

is described in terms of Erlang, it is in fact completely general and can be applied to other languages, too.

Starting from an idiomatic Erlang implementation of the EMAS, where every agent has been implemented as individual lightweight Erlang process, we have shown that the efficiency of such system may be significantly improved, by applying program shaping refactorings to introduce algorithmic skeletons. The described effect was especially visible for Rastrigin function optimisation (where the fitness function had a very low, linear cost, so communication-related issues could be significantly improved). For LABS optimisation, the cost of the fitness function (approximately quadratic) resulted in achieving similar speedups for all the tested configurations. In the best case, we have achieved super-linear speedups over the original sequential algorithm of over $70\times$ on a 64-core machine.

For future work, we intend to apply our approach to other use cases and to further evaluate its effectiveness. It would be interesting to apply it to the Erlang Dialyzer, for example, and to compare our techniques with those that have been applied manually. We also intend 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] Brown, C.—Janjic, V.—Goli, M.—Hammond, K.—McCall, J.: Bridging the Divide: Intelligent Mapping for the Heterogeneous Programmer. High-Level Programming for Heterogeneous and Hierarchical Parallel Systems, 2013.

[2] Brown, C.—Danelutto, M.—Hammond, K.—Kilpatrick, P.—Elliott, A.: Cost-Directed Refactoring for Parallel Erlang Programs. International Journal of Parallel Programming, 2013, pp. 1–19.

[3] Brown, C.—Janjic, V.—Hammond, K.—Schöner, H.—Idrees, K.—Glass, C.: Agricultural Reform: More Efficient Farming Using Advanced Parallel Refactoring Tools. 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2013.

[4] Brown, C.—Li, H.—Thompson, S.: An Expression Processor: A Case Study in Refactoring Haskell Programs. In: Page, R. (Ed.): Eleventh Symposium on Trends in Functional Programming, May 2010, 15 pp.

[5] Brown, C.—Thompson, S.: Clone Detection and Elimination for Haskell. Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '10). ACM, New York, NY, USA, 2010, pp. 111–120.

[6] Burstall, R. M.—Darlington, J.: A Transformation System for Developing Recursive Programs. Journal of the ACM (JACM), Vol. 24, 1977, No. 1, pp. 44–67.

[7] Byrski, A.—Kisiel-Dorohinicki, M.: Immunological Selection Mechanism in Agent-Based Evolutionary Computation. In: Kłopotek, M. A., Wierzchoń, S. T., Trojanowski, K. (Eds.): Intelligent Information Processing and Web Mining. Proceedings

of the International IIS: IIPWM '05 Conference, Gdansk, Poland. Advances in Soft Computing, Springer Verlag, 2005, Vol. 31, pp. 411–415.

[8] Byrski, A.—Kisiel-Dorohinicki, M.—Nawarecki, E.: Agent-Based Evolution of Neural Network Architecture. In: Hamza, M. (Ed.): Proceedings of the IASTED International Symposium: Applied Informatics. IASTED/ACTA Press, 2002.

[9] Byrski, A.—Schaefer, R.: Formal Model for Agent-Based Asynchronous Evolutionary Computation. 2009 IEEE Congress on Evolutionary Computation, May 2009, pp. 78–85.

[10] Byrski, A.—Schaefer, R.—Smołka, M.: Asymptotic Guarantee of Success for Multi-Agent Memetic Systems. Bulletin of the Polish Academy of Sciences – Technical Sciences, Vol. 61, 2013, No. 1.

[11] Byrski, A.: Tuning of Agent-Based Computing. Computer Science, Vol. 14, 2013, No. 3, pp. 491–512.

[12] Byrski, A.—Drezewski, R.—Siwik, L.—Kisiel-Dorohinicki, M.: Evolutionary Multi-Agent Systems. The Knowledge Engineering Review, Vol. 30, 2015, No. 3, pp. 171–186.

[13] Byrski, A.—Kisiel-Dorohinicki, M.: Immune-Based Optimization of Predicting Neural Networks. In: Sunderam, V. S., van Albada, G. D., Sloot, P. M. A., Dongarra, J.: Computational Science – ICCS 2005. Proceedings of 5th International Conference, Atlanta, GA, USA, May 22–25, 2005, Part III. Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 3516, 2005, pp. 703–710.

[14] Cantú-Paz, E.: A Survey of Parallel Genetic Algorithms. Calculateurs Parallèles Réseaux et Systèmes Répartis, Vol. 10, 1998, No. 2, pp. 141–171.

[15] Cetnarowicz, K.—Kisiel-Dorohinicki, M.—Nawarecki, E.: The Application of Evolution Process in Multi-Agent World (MAW) to the Prediction System. In: Tokoro, M. (Ed.): Proceedings of the 2nd International Conference on Multi-Agent Systems (ICMAS '96), AAAI Press, 1996, pp. 26–32.

[16] Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. Parallel Computing, Vol. 30, 2004, No. 3, pp. 389–406.

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

[18] Dig, D.: A Refactoring Approach to Parallelism. IEEE Software, Vol. 28, 2011, pp. 17–22.

[19] Digalakis, J. G.—Margaritis, K. G.: An Experimental Study of Benchmarking Functions for Genetic Algorithms. 2000 IEEE International Conference on Systems, Man, and Cybernetics, 2000, Vol. 5, pp. 3810–3815.

[20] Dobrowolski, G.—Kisiel-Dorohinicki, M.—Nawarecki, E.: Some Approach to Design and Realisation of Mass Multi-Agent Systems. In: Schaefer, R., Sedziwy, S. (Eds.): Advances in Multi-Agent Systems. Jagiellonian University, 2001.

[21] Fields, D. K.—Saunders, S.—Belyaev, E.: IntelliJ IDEA in Action. Manning, 2006.

[22] Fluet, M.—Rainey, M.—Reppy, J.—Shaw, A.—Xiao, Y.: Manticore: A Heterogeneous Parallel Language. Proceedings of the 2007 Workshop on Declarative As-

pects of Multicore Programming (DAMP '07), ACM, New York, NY, USA, 2007, pp. 37–44.

[23] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[24] Gallardo, J. E.—Cotta, C.—Fernández, A. J.: Finding Low Autocorrelation Binary Sequences with Memetic Algorithms. Applied Soft Computing, Vol. 9, 2009, No. 4, pp. 1252–1262.

[25] González-Vélez, H.—Leyton, M.: A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. Software – Practice and Experience, Vol. 40, 2010, No. 12, pp. 1135–1160.

[26] Groot, S.—Harmen, L. A.—van der Spek, E. M.—Bakker, H.—Wijshoff, A. G.: The Automatic Transformation of Linked List Data Structures. Proceedings of PACT, 2007.

[27] Hammond, K.—Aldinucci, M.—Brown, C.—Cesarini, F.—Danelutto, M.—González-Vélez, H.—Kilpatrick, P.—Keller, R.—Rossbory, M.—Shainer, G.: The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. Formal Methods for Components and Objects, Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 7542, 2013, pp. 218–236.

[28] Horpácsi, D.: Extending Erlang by Utilising RefactorErl. Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop, 2013.

[29] Horváth, Z.: Refactoring Rules. Technical report, ELTE-Soft, July 2014, Deliverable 4.5 of the ParaPhrase Project.

[30] Janjic, V. et al.: RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications. 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), February 2016.

[31] Hammond, K.: Discovering Parallel Pattern Candidates in Erlang. Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang (Erlang '14), ACM, New York, NY, USA, 2014, pp. 13–23.

[32] Kozsik, T.—Csörnyei, Z.—Horváth, Z.—Király, R.—Kitlei, R.—Lövei, L.—Nagy, T.—Tóth, M.—Víg, A.: Use Cases for Refactoring in Erlang. Central European Functional Programming School, Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 5161, 2008, pp. 250–285.

[33] Krzywicki, D.—Turek, W.—Byrski, A.—Kisiel-Dorohinicki, M.: Massively-Concurrent Agent-Based Evolutionary Computing. Journal of Computational Science, Vol. 11, November 2015, pp. 153–162.

[34] Li, H.—Thompson, S.: Tool Support for Refactoring Functional Programs. Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '08), ACM, New York, NY, USA, 2008, pp. 199–203.

[35] Li, H.—Thompson, S.: A Domain-Specific Language for Scripting Refactorings in Erlang. Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE '12). Fundamental Approaches to Software Engineer-

ing, Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 7212, 2012, pp. 501–515.

[36] Li, H.—Thompson, S.: Safe Concurrency Introduction Through Slicing. Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation (PEPM '15), ACM, New York, NY, USA, 2015, pp. 103–113. 2015, ACM.

[37] Li, H.—Thompson, S.—Orosz, G.—Tóth, M.: Refactoring with Wrangler, Updated: Data and Process Refactorings, and Integration with Eclipse. Proceedings of the 7th ACM SIGPLAN Workshop on Erlang (Erlang '08), ACM, New York, NY, USA, 2008, pp. 61–72.

[38] Li, H.—Thompson, S. J.—Reinke, C.: The Haskell Refactorer, HaRe, and Its API. Electronic Notes in Theoretical Computer Science, Vol. 141, 2005, No. 4, pp. 29–34.

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

[40] Marlow, S.—Newton, R.—Peyton Jones, S.: A Monad for Deterministic Parallelism. Proceedings of the 4th ACM Symposium on Haskell (Haskell '11), ACM, New York, NY, USA, 2011, pp. 71–82.

[41] McCool, M.—Robison, A.—Reinders, J.: Structured Parallel Programming. Morgan Kaufmann, 2012.

[42] Mens, T.—Tourwé, T.: A Survey of Software Refactoring. IEEE Transactions on Software Engineering, Vol. 30, 2004, pp. 126–139.

[43] Opdyke, W. F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, 1992.

[44] Partsch, H.—Steinbrüggen, R.: Program Transformation Systems. Computing Surveys, ACM, Vol. 15, 1983, No. 3, pp. 199–236.

[45] Schaefer, R.—Byrski, A.—Smolka, M.: The Island Model as a Markov Dynamic System. International Journal of Applied Mathematics and Computer Science, Vol. 22, No. 4, pp. 971–984.

[46] Siwik, L.—Dreżewski, R.: Agent-Based Multi-Objective Evolutionary Algorithms with Cultural and Immunological Mechanisms. In: Wellington Pinheiro dos Santos (Ed.): Evolutionary Computation, In-Teh, 2009, pp. 541–556.

[47] Skillicorn, D.: Foundations of Parallel Programming. Cambridge University Press, New York, NY, USA, 1995.

[48] Sultana, N.—Thompson, S.: Mechanical Verification of Refactorings. Workshop on Partial Evaluation and Program Manipulation, ACM SIGPLAN, January 2008, pp. 182–196.

[49] Trinder, P. W.—Hammond, K.—Loidl, H.-W.—Peyton Jones, S. L.: Algorithm + Strategy = Parallelism. Journal of Functional Programming, Vol. 8, 1998, No. 1, pp. 23–60.

[50] Uhruski, P.—Grochowski, M.—Schaefer, R.: A Two-Layer Agent-Based System for Large-Scale Distributed Computation. Computational Intelligence, Vol. 24, 2008, No. 3, pp. 191–212.

[51] Wooldridge, M. J.: An Introduction to Multiagent Systems. John Wiley & Sons, 2009.

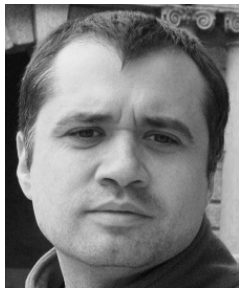
[52] Wróbel, K.—Torba, P.—Paszyński, M.—Byrski, A.: Evolutionary Multi-Agent Computing in Inverse Problems. Computer Science, Vol. 14, 2013, No. 3, pp. 367–383.

Adam Barwell is a Ph.D. student at the University of St Andrews. He is interested in programming languages, static analysis, refactoring, and parallel programming.

Christopher Brown received his Ph.D. degree from the University of Kent in 2009. He now works as Senior Postdoctoral-Research Fellow at the University of St Andrews, where his research focuses on pioneering advanced refactoring techniques for multi-core systems.

Kevin Hammond is Full Professor of Computer Science at the University of St Andrews, where he leads the functional programming research group. His research interests lie in programming language design and implementation, with a focus on parallelism and real-time properties of functional languages, including modelling and reasoning about extra-functional properties. In total, he has published around 100 research papers, books and articles, and held over 20 national and international research grants, totalling around 11M of research funding. He was a member of the Haskell design committee, co-designed the Hume real-time functional language, and is co-editor of the main reference text on parallel functional programming. He currently coordinates the RePhrase project, a 3-year EU research project that aims to develop new refactoring technology targeting heterogeneous parallel architectures. He is a keen hill-walker, whisky connoisseur and enjoys early music.

Wojciech Turek received his Ph.D. degree in 2010 from the AGH University of Science and Technology in Cracow. He works in the area of multi-robot systems, multi-robot planning, autonomous and agent-based systems, concurrent and parallel programming, mostly in functional languages.



Aleksander Byrski received his Ph.D. in 2007 and his D.Sc. (habilitation) in 2013 from the AGH University of Science and Technology in Cracow, Poland. He works as Assistant Professor at the Department of Computer Science of AGH-UST. His research focuses on multi-agent systems, biologically-inspired computing and other soft computing methods.