Int J Parallel Prog (2018) 46:4–22
https://doi.org/10.1007/s10766-017-0503-4

The Missing Link! A New Skeleton for Evolutionary Multi-agent Systems in Erlang

Jan Stypka[1] · **Wojciech Turek**[1] · **Aleksander Byrski**[1] · **Marek Kisiel-Dorohinicki**[1] · **Adam D. Barwell**[2] · **Christopher Brown**[2] · **Kevin Hammond**[2] · **Vladimir Janjic**[2]

Received: 4 October 2016 / Accepted: 11 April 2017 / Published online: 26 April 2017

Abstract Evolutionary multi-agent systems (EMAS) play a critical role in many artificial intelligence applications that are in use today. In this paper, we present a new generic skeleton in Erlang for parallel EMAS computations. The skeleton enables us to capture a wide variety of concrete evolutionary computations that can exploit the same underlying parallel implementation. We demonstrate the use of our skeleton on two different evolutionary computing applications: (1) computing the minimum of the Rastrigin function; and (2) solving an urban traffic optimisation problem. We show that we can obtain very good speedups (up to 142.44$\times$ the sequential performance)

✉ Christopher Brown
cmb21@st-andrews.ac.uk

Jan Stypka
janstypka@gmail.com

Wojciech Turek
wojciech.turek@agh.edu.pl

Aleksander Byrski
olekb@agh.edu.pl

Marek Kisiel-Dorohinicki
doroh@agh.edu.pl

Adam D. Barwell
adb23@st-andrews.ac.uk

Kevin Hammond
kh8@st-andrews.ac.uk

Vladimir Janjic
vj32@st-andrews.ac.uk

1 AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland

2 School of Computer Science, The University of St Andrews, St Andrews, UK

on a variety of different parallel hardware, while requiring very little parallelisation effort.

Keywords Multi-core programming · Erlang · Agent-based computing · Metaheuristics · Many-core programming · Algorithmic skeletons

1 Introduction

Evolutionary Multi-Agent Systems [9] (EMAS) are a critical part of many modern artificial intelligence systems. The computations that they perform are usually very expensive and, since they involve individual autonomous entities (agents) with no central authority involved in their operation, they should be highly amenable to parallelisation. Desipite this, there has been relatively little effort spent to date on developing parallel solutions for EMASs and the solutions that are available are usually tailored to a specific instance of a more general problem. At the same time, we are witnessing the emergence of many-core systems across various levels of the computing spectrum, from low-power devices targeting data-centres on a chip/cyber-physical systems all the way to high-performance supercomputers. These systems offer significant processing potential and are ideally suited to improve the performance of EMAS. However, harnessing that potential is still a huge issue, both for EMAS and for many other applications. This is mainly due to the widespread use of low-level native parallel programming models that are commonly used to program these systems (e.g. pThreads and OpenMP).

In this paper, we present a new implementation of an algorithmic skeleton for evolutionary multi-agent systems. Algorithmic skeletons are implementations of common parallel patterns, parameterised over worker functions and other implementation-specific information. A skeleton can be specialised to a specific problem by providing concrete instantiations of worker functions and other required information. Our EMAS skeleton is implemented in the functional programming language Erlang. Erlang is widely used in the telecommunications industry, but is also beginning to be used more widely for high-reliability/ highly-concurrent systems, e.g. databases [27], AOL's *Marketplace by Adtech* [31] and WhatsApp [28]. It has excellent support for concurrency and distribution, including built-in fault tolerance. The latter is critical for long-running computations such as evolutionary computations. Functional programming approaches naturally provide high-level abstractions through e.g. higher-order functions and Erlang was therefore an ideal choice for developing a new generic parallel skeleton.

The specific research contributions of this paper are:

1. We design and implement three versions of a new *domain-specific parallel skeleton* for Evolutionary Multi-Agent Systems (EMAS) using Erlang;
2. We implement two realistic evolutionary computing use cases that use our skeleton; and,
3. We evaluate these use cases on a number of different parallel architectures, ranging from small-scale low-power systems to large-scale high-performance multicore

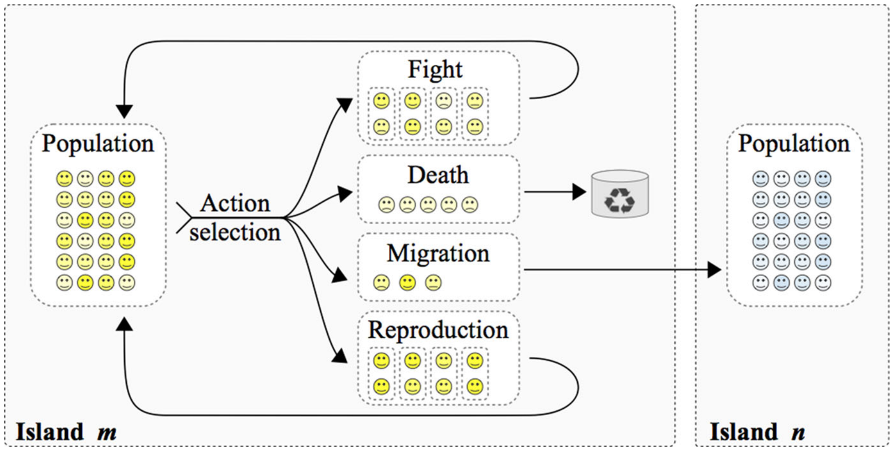


Fig. 1 The general structure of an evolutionary multi-agent system

machines, demonstrating that we can obtain very good performance improvements of up to a factor of 142 over the sequential implementations.

2 Evolutionary Multi-agent Systems and Parallel Skeletons

In Agent-oriented programming (AOP), the construction of the software is based on the concept of specialised, independent, autonomous software *agents*. There are many different agent-based programming frameworks, ranging from general-purpose ones, like RePast [25], MASON [23], MadKit [19], JADE [6] to more specialised ones, like ParadisEO [10] or AgE [26]. Some of these frameworks leverage existing multi-core architectures (e.g. ParadisEO). However, their scalability to larger systems has not been explored. In this paper, we focus on applications where agents use *evolutionary algorithms* in their operation, most frequently for learning/reasoning or coordination of some group activity.

Evolutionary computations [5,17] use the idea of mimicking the mechanisms underlying biological evolution to solve complex adaptation and optimisation problems. They work on a population of individuals, each of which represents a point in the search space of potential solutions to a given problem. The population undergoes subsequent modification steps by means of randomised genetic operations that model recombination, mutation and selection. Following initialisation, the algorithms loop through these operators until some termination criterion is satisfied. Each of these cycles is called a *generation*. *Evolutionary Multi-Agent Systems* (EMASs) are a hybrid meta-heuristic that combines multi-agent systems with evolutionary algorithms [8,9] (see Fig. 1). In a multi-agent system, no global knowledge is available to individual agents; agents remain autonomous and no central authority is involved in making their decisions. Therefore, in contrast to traditional evolutionary algorithms, in an evolutionary computing system, *selective pressure* (essentially, a method for selecting which members of the population survive and move to the next generation) 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 EMAS, emergent selective pressure is achieved by giving agents a single non-renewable resource called energy. Agents with high energy are more likely to reproduce whereas agents with low energy are more likely to die. The algorithm is designed to transfer energy from better to worse agents without central control.

In a basic EMAS implementation, every agent is provided with a real-valued vector representing a potential solution to the optimisation problem, along with the corresponding fitness metric. Agents start with an initial amount of energy and meet randomly. If their energy is below a death threshold, they die. If it is above some reproduction threshold, they reproduce and yield new agents—the genotype of the children is derived from their parents using variation operators and some amount of energy is also inherited. If neither of these two conditions is met, agents fight in tournaments by comparing their fitness values. The better (winning) agents then sap energy from the worse (losing) ones [9]. The system as a whole is *stable*, since the total energy remains constant. However, the number of agents may vary in order to adapt to the difficulty of the problem—small numbers of agents with high energy or large numbers of agents with low energy, for example. The number of agents can also be altered dynamically by varying the total energy of the system. As in other evolutionary algorithms, agents can be split into separate populations, or *islands*. Such islands help preserve diversity by introducing allopatric speciation and can also execute in parallel. Information is exchanged between islands through agent migrations. Evolutionary multi-agent systems have been formally shown to be a general optimisation tool by constructing a detailed Markov-chain based model and proving its ergodicity [8].

2.1 Parallel Programming in Erlang

Erlang [3] is a strict, impure, functional programming language with built-in support for concurrency. It supports a *lightweight* threading model, where *processes* model small units of computation. The scheduling of processes is handled automatically by the Erlang Virtual Machine, which also provides basic load balancing mechanisms. A key aspect of the Erlang design is a *shared-nothing* approach, in which processes do not implicitly share state—all data is communicated via channels, using explicit *send* and *receive* primitives. It has proven to be an efficient tool for building large-scale systems for multi-core processors. Erlang is widely used in the telecommunications industry, but is also beginning to be used more widely for high-reliability/highly-concurrent systems, e.g. databases [27], AOLs Market-place by Adtech [31], and WhatsApp [28]. Most of the industrial applications written in Erlang are deployed on systems with up to 24 CPU cores. Scalability of solutions is usually provided by using clusters of multicores and running Erlang in distributed configuration.

There are few reports on using a single Erlang Virtual Machine on architectures exceeding 32 physical cores. In [4] the authors present a test suite for measuring different aspects of Erlang applications performance. The exemplary test running on a 64-core machine shows that in most cases the speedup is non-linear and it degrades for high number of cores and schedulers. The problem of Erlang Term Storage scalability on a computer with 32 physical cores have been considered in [29]. Promising results of using Erlang on a Intel Xeon Phi co-processor have been shown in [32]. Basic benchmarks show good scalability up to 60 cores, which is the number of physical cores of the co-processor. However, there are hardly any reports on scaling complex, computationally intensive Erlang applications on many-core architectures.

Our main motivation for using Erlang for evolutionary computations is to achieve scalability. While the performance on single-core machines of Erlang is still behind more mature and established languages, like C and C++, it is designed to be easily and transparently scalable across distributed systems. In the era of many-core architectures, the ability of scaling the performance with the growing number of cores/nodes will become far more important than the effectiveness of single core utilisation.

2.2 Algorithmic Skeletons

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction [13] into parameterised templates. For example, we might define a *parallel map* skeleton, whose functionality is identical to a standard *map* function, but which creates a number of Erlang *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. Details such as communication, task creation, task or data migration and scheduling 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 typical low-level approaches. A recent survey of algorithmic skeleton approaches can be found in [18].

2.3 The *Skel* Library for Erlang

In our previous work, we have developed the *Skel* [7,20] library, which defines a small set of classical skeletons for Erlang. Each skeleton operates over a stream of input values, producing a corresponding stream of results. Skel also allows simple composition of skeletons. Skel supports the following skeletons:

- `func` is a simple wrapper skeleton that encapsulates a sequential function as a streaming skeleton.
- `pipe` models a composition of skeletons $s_1, s_2, \ldots, s_n$ over a stream of inputs. Within the pipeline, each of the s_i is applied in parallel to the result of the s_{i-1}.
- `farm` models the application of the same operation over a stream of inputs. Each of the n farm *workers* is a skeleton that operates in parallel over independent values of the input stream.
- `cluster` is a data parallel skeleton, where each independent input, x_i can be partitioned into a number of sub parts, $x_1, x_2, \ldots, x_n$, that can be worked upon in parallel. A skeleton, s, is then applied to each element of the sub-stream in parallel.

- `feedback` wraps a skeleton s, feeding the results of applying s back as new inputs to s, provided they match a filter function, f.

3 A New Skeleton for Evolutionary Multi-agent Systems

At the base of our skeleton for Evolutionary Multi-Agent Systems is a skeleton for Multi-Agent Systems (MAS). Pseudocode for sequential implementation of the MAS system is given in Algorithm 1, where function with names prepended by `Env` are specific for each instance of the skeleton.

Algorithm 1 Pseudocode for Multi Agent System (MAS) skeleton

```
for i in [1..nr_islands] do
    for j in [1..nr_agents_per_island] do
        new agent[j] = Env:initialise_agent()
        add agent[j] to island[i]
    end for
end for
while not finished() do
    for all $I \in$ Islands do
        for all agent $\in I$ do
            if rand() < migration_probability then
                agent.behaviour = migration
            else
                agent.behaviour = Env:behaviour_function(agent, Env)
            end if
        end for
        Emigrants = collect_emigrants(I)
        Env:migration_function(I, Env)
        I = I $\cup$ Emigrants
    end for
end while
```

In this skeleton, a number of agents is created and divided into islands. In each iteration of the algorithm, each agent either migrates to another island or performs a behaviour function. This decision is based on some predefined probability (migration_probability). After all the agents decide which of these two functions to perform, a migration function is performed on groups of agents that end up in the same island.

EMAS skeleton is a specialisation of the MAS skeleton, with particular versions of initialisation, behaviour and migration functions. The behaviour of this skeleton was explained in Sect. 2. The pseudocode for the sequential EMAS skeleton is given in Algorithm 2. Functions with names prepended with `Prob` are specific to each problem that is solved using the skeleton. Therefore, the EMAS skeleton accepts two types of input:

- a set of parameters for the computation, including the number of agents, initial energy and details about reproduction (e.g. the maximum energy given by a parent to a child), mutation (e.g. the chance of mutation during reproduction) and recombination (e.g. the probability of recombining parent solutions during reproduction);

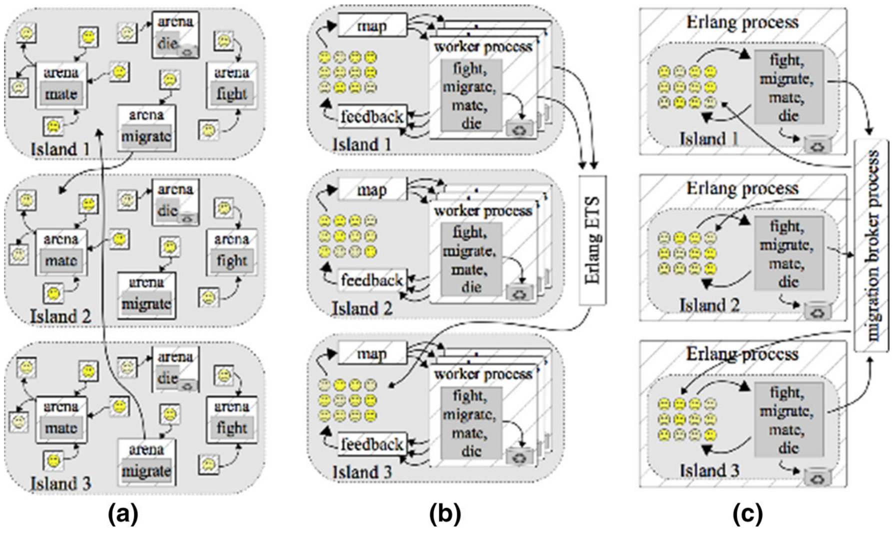


Fig. 2 Three versions of EMAS in Erlang: **a** the concurrent version with each solution represented by autonomous process; **b** the skel version with configurable number of concurrent workers; **c** the hybrid version with sequential islands computed concurrently

– problem-specific `solution`, `evaluation`, `mutation` and `recombination` functions.

We have implemented three different versions of the skeleton, which differ in the precise way in which agents are captured by Erlang processes:

a. *The Concurrent version* (Fig. 2a), where each agent is represented by a separate Erlang process. The agents use dedicated arena processes to trigger the behaviour, mutation and recombination functions. This is a fine-grained implementation of the EMAS computation, where we potentially have a very large number of very small Erlang processes;
b. *The Skel version* (Fig. 2b), which uses the *feedback* parallel pattern from the Skel pattern library (described in Sect. 2.3). Here, we set a fixed number of Erlang processes that will be created (usually matching the number of OS threads that we want to use) and then distribute agents from a population to these worker processes. When one generation finishes computing, the resulting new population is merged by the feedback process. Separate Erlang processes are created to handle the fighting, migration, reproduction and dying parts of the computation.
c. *The Hybrid version* (Fig. 2c), which represents an optimised version of the *Skel* version (described below). The population of agents is split into islands and every island is contained in a separate Erlang process. Within the process, the `fight`, `reproduce` and `die` functions are done sequentially, and a separate dedicated Erlang process handles migration of agents. This version represents coarse-grained implementation of the EMAS computation, with a small number of potentially expensive Erlang processes than in the previous two versions;

Algorithm 2 Pseudocode for specialisation of MAS skeleton into EMAS skeleton

```
function INITIALISE_AGENT
    Prob:solve()
end function

function BEHAVIOUR_FUNCTION(ReprodThr,Energy)
    if Energy == 0 then return death
    else if Energy > ReprodThr then return reproduction
    else  return fight
    end if
end function

function MEETING_FUNCTION(Agents, Behaviour, Env)
    if Behaviour == death then return []
    else if Behaviour == reproduction then do_reproduction(Agents, Env)
    else if Behaviour == fight then do_fight(Agents, Env)
    end if
end function

function DO_FIGHT(Agents) Agents={{Agent1, Ev1, En1},{Agent2, Ev2, En2}}
    if Ev1>Ev2 then TransferEng = En2
    else  TransferEng = En1
    end if
    return {{Agent1, Ev1, En1 + TransferEng}, {Agent2, Ev2, En2 - TransferEnf}}
end function

function DO_REPRODUCTION(Agents)
    if Agents={{Agent1, Ev1, En1}} then
        NewAgent = reproduction(Agent1)
        NewEval = Prob:evaluation(Agent1)
        TransEng = min(reproduction_transfer, En1)
        return [{Agent1, Ev1, En1 - TransEn},{NewAgent, NewEval, TransEn}]
    else if Agents={Agent1, Agent2} then
        [NewAg1, NewAg2] = reproduction(Agent1,Agent2)
        [NewEv1, NewEv2] = Prob:evaluation(Agent1, Agent2)
        Transfer energy from old to new agents
        return [Agent1, Agent2, NewAg1, NewAg2]
    end if
end function

function REPRODUCTION(Agents)
    if Agents={Agent} then
        if rand() < mutation_chance then Prob:mutation(Agent)
        else  return Agent
        end if
    else if Agents={ {Agent1, Agent2} } then
        if rand() < recombination_chance then
            [R1,R2] = Prob:recombination(Agent1, Agent2)
        else  [R1,R2] = [Agent1, Agent2]
        end if
        ...Probabilistic mutation of Agent1 and Agent2, similarly to above...
    end if
end function
```

```
-module(traffic_bin_ops).

-behaviour(emas_genetic_ops).

-export ([evaluation/2, mutation/2, recombination/3,
          solution/1, config/0]).

-include_lib("emas/include/emas.hrl").

%% @doc Generates a random solution, a list of 4-bit tuples
solution(#sim_params{problem_size = ProblemSize}) ->
    crypto:rand_bytes(ProblemSize).


%% @doc Evaluates a given solution and returns a fitness value
evaluation(Binary, #sim_params{extra = Data}) ->
    Solution = [{B1, B2, B3, B4} ||
                   <<B1:1, B2:1, B3:1, B4:1, _:4>> <= Binary],
    Fitness = evaluation:evaluate_solution(Solution, Data),
    float(Fitness).


%% @doc Crossover recombination in a random point
recombination(Sol1, Sol2,
              #sim_params{problem_size = ProblemSize}) ->
    CutPoint = random:uniform(ProblemSize),
    <<S1a:CutPoint/binary, S1b/binary>> = Sol1,
    <<S2a:CutPoint/binary, S2b/binary>> = Sol2,
    {<<S1a/binary, S2b/binary>>, <<S2a/binary, S1b/binary>>}.


%% @doc Mutates (flips) genes at random indexes
mutation(Solution, #sim_params{mutation_rate = MutationRate}) ->
    << << case random:uniform() < MutationRate of
             true -> flip_gene(Gene);
             false -> Gene
         end >> || <<Gene>> <= Solution >>.
```

Fig. 3 Example of the use of the EMAS skeleton (traffic)

Since it is most coarse-grained and has the least synchronisation between processes, we expect *Hybrid* to give the best performance of these three versions.

3.1 Use of the EMAS Skeleton

As we have mentioned in the previous section, in order to use the EMAS skeleton on a particular problem, the programmer needs to supply problem-specific versions of the solution, recombination, evaluation and mutation functions, together possibly with simulation parameters, if default values are not acceptable. Everything else, including the main function (`start()`) is provided by the skeleton. Figure 3 shows an example of an input file with these functions, with some auxiliary functions missing. This example (Traffic optimisation) was used in the Sect. 4 to evaluate the EMAS skeleton.

Table 1 Experimental platforms

	pi	*titanic*	*zeus*	*power*	*phi*
Arch	Arm	AMD	AMD	IBM	Intel
Proc	Cortex-A7	Opteron 6176	Opteron 6276	Power8	Xeon Phi 7120
Cores	4	24	64	20	61
Threads	4	24	64	160	244
Freq.	900 MHz	2.3 GHz	2.3 GHz	3.69 GHz	1.2 GHz
L2 Cache	256 KB	24 × 512 KB	32 × 2 MB	20 × 512 KB	61 × 512KB
L3 Cache	–	4 × 6 MB	4 × 8 MB	20 × 8 MB	–
RAM (GB)	1	32	256	256	16

4 Evaluation

We have evaluated the performance of our EMAS skeleton on two different optimisation examples that typify agent-based evolutionary computations:

1. Finding the minimum of the Rastrigin function, a popular global optimisation benchmark [15];
2. A complex problem of urban traffic optimisation, a multi-variant simulation [21] that is applied to the mobile robotics domain. Our model anticipates possible situations on the road, preparing plans to deal with certain situations and apply these plans when certain traffic conditions arise.

Since the algorithms are bounded by time, we measure the average rate of reproductions that the algorithm achieves per second, representing its throughput. We also measure the increase in throughput as the number of cores increases. This corresponds to an increase in the application's performance (speedup). We ran each experiment a total of 10 times, with each experiment set to take 5 min. The number of islands that the algorithm is able to use is set to be the number of cores. We prioritised the evaluation of the hybrid configuration, since it has proven itself to be the best of the three parallel configurations, but have included other configurations in our results where available. For all experiments, we have used version 18 of the Erlang system, except where otherwise highlighted. In order to evaluate the EMAS skeleton on a number of different architectures and in different settings, our experiments have been conducted on five different systems, ranging from low-power small-scale ARM systems (*pi*) through more powerful ×86 (*titanic*, *zeus*) and (*power*) OpenPower multicores to a standalone Intel Xeon Phi many-core accelerator. Full details of the systems that we have used are given in Table 1. We focus on measuring the scalability of the system instead of the sequential performance. We have carried out preliminary tests comparing existing models to a purely sequential version, however the performance differences were insignificant in comparison to benefits introduced by parallelism.

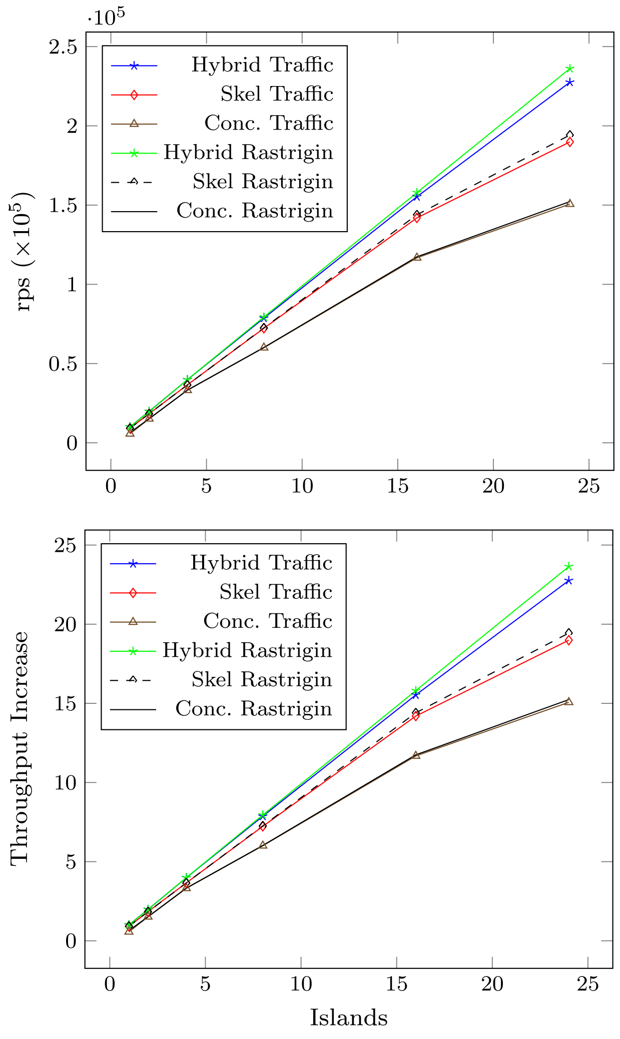


Fig. 4 Reproductions per second (*rps*, *above*) and throughput increase (*below*) for traffic and rastrigin on *titanic*

4.1 *titanic* Results

The *titanic* system is an example of a medium-scale parallel server, with 24 cores and 32 GB of RAM. Figure 4 shows the number of reproductions per second (*rps*) and the increases in throughput for Traffic and Rastrigin on this machine. For both applications, we observe that the Hybrid version achieves almost linear increases in throughput (and, therefore, performance). For Rastrigin, the Hybrid version achieves 236,140 *rps* on 24 cores versus 10,038 *rps* on one core, yielding 23.52× the throughput (or 23.64× the base sequential throughput). For Traffic, the Hybrid version on *titanic* achieves 227,509 *rps* on 24 cores versus 10,059 *rps* on one core, yielding 22.62×

the throughput (or 22.76× the base sequential throughput of 9996 *rps*). The Skel and Concurrent versions initially also perform well, but start to tail off after about 16 cores (islands) for both applications. This is consistent with other applications we have run on this machine, and is probably due to cache contention. Although the Hybrid version is always best, the untuned Skel version always performs better than the Concurrent version. This shows the benefit of reducing the number of Erlang processes by grouping the computations that multiple agents perform into a single process.

4.2 *zeus* Results

The *zeus* system is an example of a larger-scale multicore system, with 64 cores and 256 GB of RAM. Figure 5 shows the *rps* and corresponding throughput increase for all the tested versions. Here, Rastrigin achieves 349,226 *rps* on 64 cores versus 5752 *rps* on one core, for a throughput increase of 60.71× the sequential version. Traffic achieves 94,950 *rps* on 64 cores versus 1473 *rps* on one core, for a throughput increase of 64× the sequential version. There is clearly a tail-off in throughput increase for Traffic at 32 and 48 cores, but further experiments are required to explain the reasons for this, given the clear improvement with 64 cores. A slight dip can also be observed for Rastrigin at 32 cores (27.8× throughput increase), but this has recovered at 48 cores (46.13× throughput increase). In comparison with *titanic*, the raw performance per core is lower for both applications (despite a nominally similar processor architecture), but *zeus* delivers higher total throughput.

4.3 *phi* Results

Our motivation for considering the *phi* system was to evaluate the performance of our skeletons (and, generally, of the Erlang runtime system) on a many-core accelerator. We have therefore focused on using just the accelerator, without also using the multi-core host system. Figure 6 shows the reproductions per second and throughput increase for the Hybrid version of the two use cases. We were unfortunately unable to run experiments with the Concurrent and Skel versions on the Xeon Phi as detailed adaptation of the Concurrent and Skel code for Xeon Phi was required, and we wanted to retain code compatibility among the significant number of different processors we used during the experiments. This is left as an area of future work. Note that the system has 61 physical cores, but can run 244 simultaneous threads via 4 way hyper-threading. Due to the sharing of resources between multiple threads, we do not expect to see 244 times increase in throughput on this architecture. Indeed, we can clearly see that the performance improvement decreases above 61 cores and tails off when more than 122 threads are used. Even so, we were able to achieve very good results, improving performance more than 100 times (compared to the sequential version) when 244 threads were used for both applications, and achieving excellent improvements up to 61 cores for both applications. Here, Rastrigin achieves 188,240 *rps* on 244 cores versus 1681 *rps* on one core, for a throughput increase of 111.96× the sequential version. Traffic

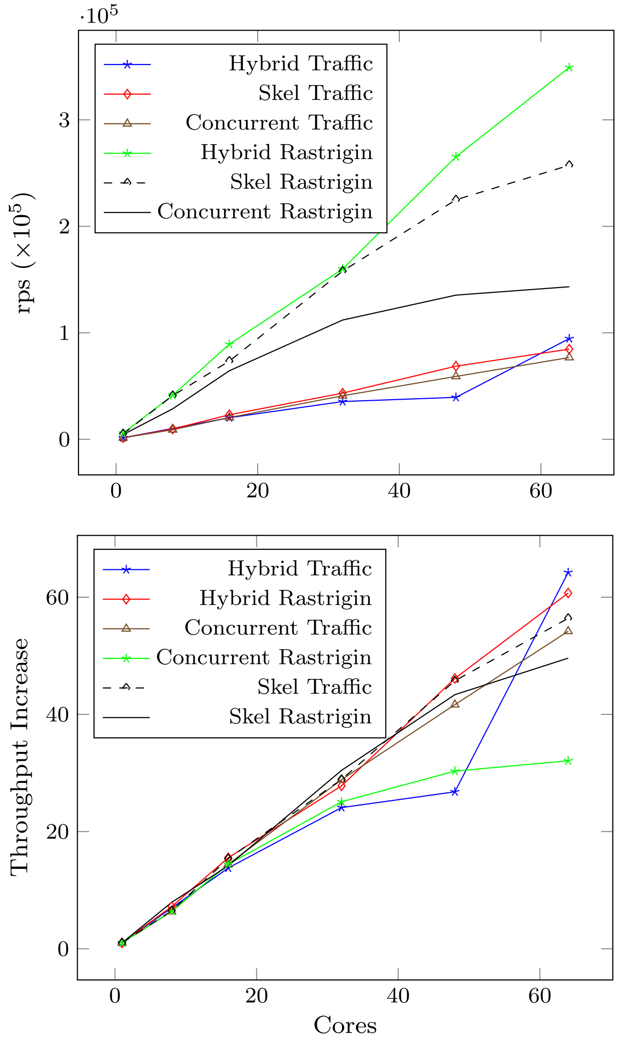


Fig. 5 Reproductions per second (*rps*, *above*) and throughput increase (*below*) for traffic and rastrigin on *zeus*

achieves 42,902 *rps* on 244 cores versus 302 *rps* on one core, for a throughput increase of 142.44× the sequential version.

The number of islands in the experiments on *zeus* and *phi* was fixed at the level of the number of available logical cores. This approach lead to testing exactly the same algorithm on different number of cores, which has been adjusted by setting the number of schedulers of the Erlang VM. The side-effect of this approach was that the initialisation of the islands took proportionally more time with the decrease of the number of cores, influencing the measured throughput. This overhead caused the super-linearity effect on *phi*—the initialisation time was shorter for greater number of cores.

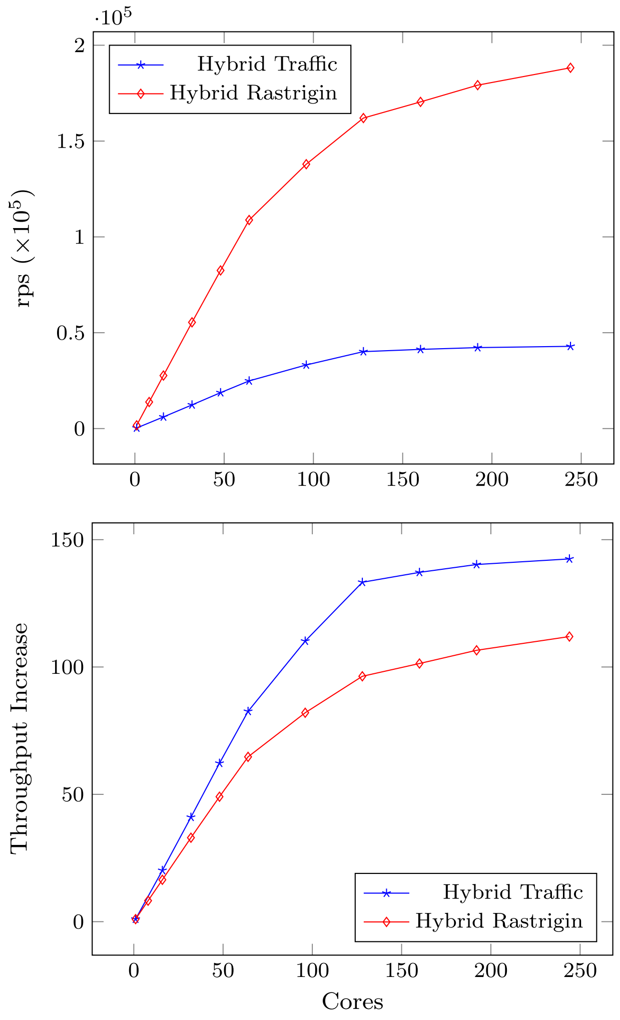


Fig. 6 Reproductions per second (*rps*, *above*) and throughput increase (*below*) for traffic and rastrigin on *phi*

4.4 Preliminary *power* Results

The *power* system represents the IBM Power8 architecture, which is designed to allow a highly multi-threaded chip implementation. Each core is capable of handling 8 hardware threads, which gives a total of 160 threads on a system with 20 physical cores. At 3.69Ghz, the clock frequency is also noticeably higher than for the other systems that we have considered. However, as with the Xeon Phi, resources are shared between multiple threads and we therefore do not see a linear increase in throughput with the number of cores (islands). Due to problems with running *Rastrigin*, we only present the results for the *Traffic* use case (Fig. 7). Since the latest version of Erlang (Erlang 18)

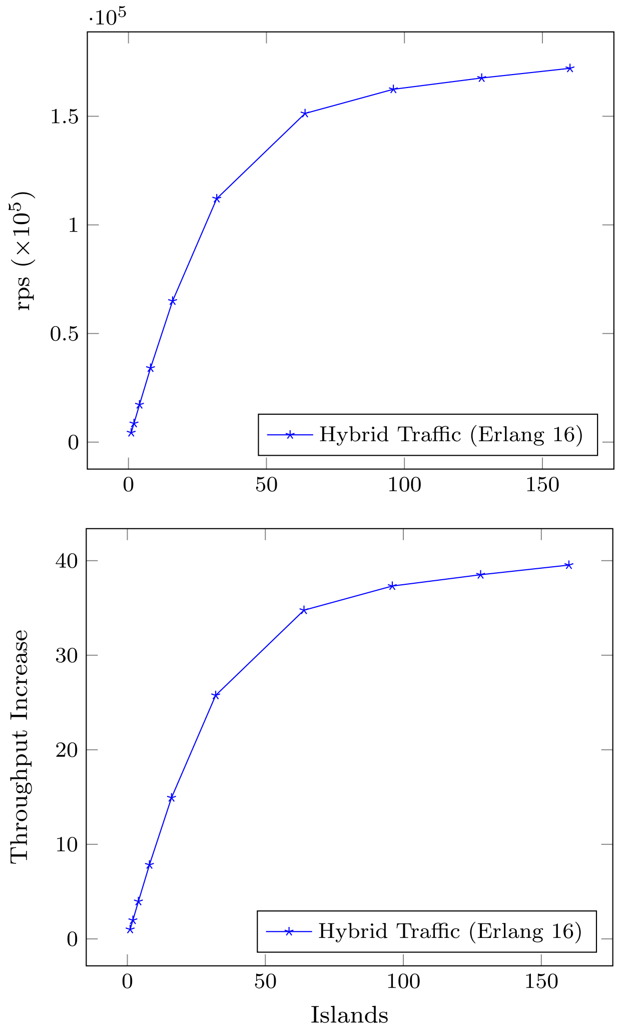
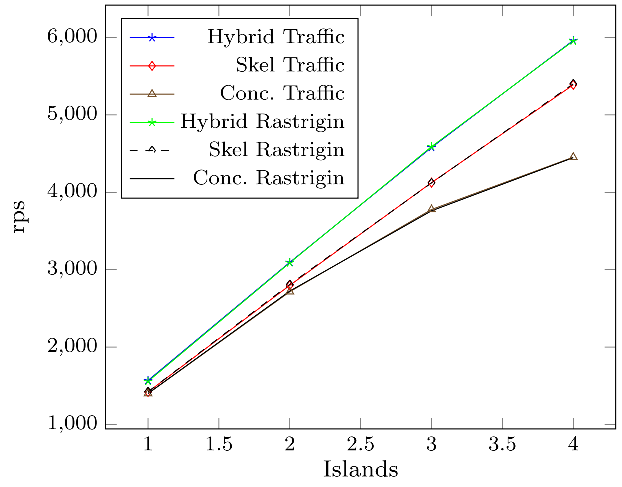


Fig. 7 Reproductions per second (*rps*, *above*) and throughput increase (*below*) for traffic on *power* (Erlang 16)

is currently unsupported on the OpenPower architecture, our experiments use an older version, Erlang 16. Our tests on other systems suggest that this dramatically lowers the absolute throughput of the system (by about a factor of 3). Overall, we achieve 172,078 *rps* on 160 cores versus 1,681 *rps* on one core, for a throughput increase of 39.53× the sequential version. We observe very good scaling up to about 40 threads (2 per physical core), achieving throughput improvement of 25.75 on 32 cores and 34.76 on 64 cores. Beyond that, as with the *phi* system, performance improvements tail off rapidly, but smoothly. Currently we were only able to port the *hybrid* versions of the EMAS skeleton to *power*, with other versions planned as future work.



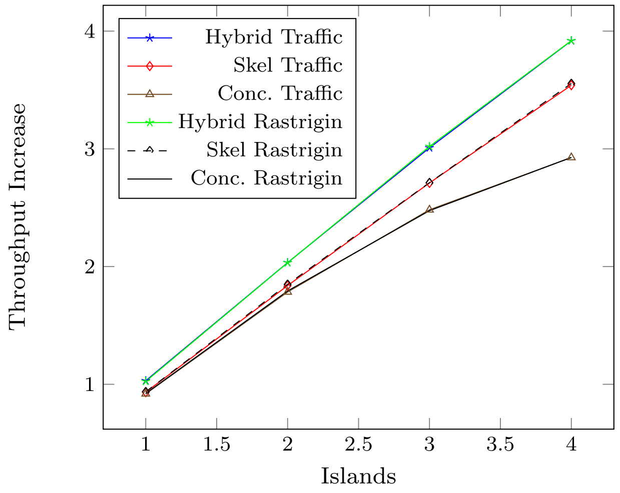


Fig. 8 Reproductions per second (*rps*, *above*) and throughput increase (*below*) for traffic and rastrigin on *pi*

4.5 *pi* Results

The *pi* system is a quad-core Raspberry Pi 2, model B. We have chosen it as an example of a low-power parallel architecture, that is intended to be used in e.g. high-end embedded systems. While we expect the absolute performance of this system to be significantly lower than the other, heavyweight, parallel systems we have studied, it is still interesting to in evaluating how well our skeletons perform on such a system. Figure 8 shows the results for *pi*. We have considered all three versions of the EMAS skeleton. As for *titanic*, the Hybrid version gives the best performance, and the *Skel* version outperforms the *Concurrent* version. For Traffic, the Hybrid version on *titanic* achieves 5962 *rps* on 4 cores versus 1568 *rps* on one core, yielding 3.92× the throughput. For Rastrigin, the Hybrid version achieves almost identical results of 5954 *rps* on 4 cores versus 1556 *rps* on one core, also yielding 3.92× the throughput. We observe similar results for the other two versions: with throughput improvements of 3.54/3.54 for the Skel version on Traffic/Rastrigin (5387/5401 *rps*), and 2.93 for the Concurrent version (4453/4447 *rps*). Although, as expected, the absolute peak performance (about 6000 reproductions per second) was orders of magnitude weaker than on other systems; however, it is worth noting that Pi power consumption is just a fraction of these of heavyweight servers, so it gives the best performance-per-watt ratio of all systems we tested.

5 Related Work

Algorithmic skeletons [13] have been the focus of much research since the 1980s, with a number of skeleton libraries being produced in a range of languages [1,18]. These include our own Skel library [7], which is currently the only available skeleton library for Erlang. A recent alternative approach is to define an embedded domain-specific language (EDSL) [12,22], This can provide a structured approach to parallelism, similar to our approach using skeletons. Skeletons, however, have the advantage of representing *language-independent* patterns of parallelism, that can more easily be transferred to other language settings. Other high-level approaches that hide the low-level parallel mechanics from the programmer include: futures [16], evaluation strategies [30], and parallel monads [24].

There have been a few attempts to parallelise agent systems. For example Al-Jaroodi *et al.* have presented an implementation of a Java-based agent-oriented system, achieving linear scalability up to 8 cores [2], and Cicirelli and Nigri have achieved similar scale-up in a system that is aimed at simulation [11]. Cosenza *et al.* describe an implementation of a C++/MPI scalable simulation platform for spatial simulations of particle movement, achieving linear scalability up to 64 cores [14]. Finally, Cahon *et al.* have achieved linear speedup on up to 10 cores for a C++ implementation of ParadisEO [10]. All of these systems were tested on either small scale parallel systems, or are tailored to a specific evolutionary computing application. In contrast, our skeleton is sufficiently general to cover a wide class of evolutionary computing applications, and we have also demonstrated its scalability on systems with up to 244 hardware threads, achieving maximum throughput improvements of up to 142.44× the sequential version.

6 Conclusions and Future Work

Evolutionary multi-agent systems (EMAS) are a very important component of many artificial intelligence systems. In this paper, we have described the design and implementation of a new domain-specific skeleton for evolutionary multi-agent systems (EMAS) in Erlang. By implementing a skeleton for a widely used computational pattern, we enable easy parallelisation of a large class of applications, where programmer is required to supply only problem-specific sequential components, and all parallelism is handled by the skeleton implementation. We also enable easy cross-platform portability. We have developed three different versions of the skeleton—the *Concurrent* version that is based on Erlang processes, the *Skel* version that is based on the Skel library of parallel skeletons, and the *Hybrid* tuned version of the *Skel* version. We evaluated these different versions of the skeleton on two different evolutionary computing applications: (1) finding the minimum of the Rastrigin function; and (2) an urban traffic optimisation. These applications come from different domains and were adapted to use the same skeleton. We have achieved very good improvements in performance on a number of different architectures, ranging from small-scale low-power multi-core systems (a quad-core Raspberry Pi) to larger multi-core servers (a 64-core AMD Opteron) and a many-core accelerator (Intel Xeon Phi) with very little coding effort. This showed the scalability and adaptability of our skeleton implementation in a number of different deployment settings. As expected, the best implementation was consistently the Hybrid version. Our results show that we can achieve throughput improvements for this version of up to a factor of 142.44 compared to the sequential version (for the Xeon Phi), with near-linear improvements on the servers and Raspberry Pi. Overall, the 64-core multi-core system (*zeus*) gave the best result for Rastrigin (349,226 *rps*). However, rather surprisingly, the 24-core server (*titanic*) achieves the best result for Traffic (227,509 *rps* vs. 94,950 *rps*). Given their pricing relative to the other architectures, the results for the Xeon Phi and Raspberry Pi are very creditable. Although the OpenPower results are a little disappointing in absolute terms, this is largely because an older version of Erlang was used. If the newest version of Erlang was available, we would expect this architecture to deliver the best absolute results. Finally, we had not expected the architectures to have such widely varying performance. The fact that our skeletons (notably the Hybrid version) dealt well and, on the whole, predictably with all of these architectures is a positive result.

Our main line of future work is to adapt our skeleton to distributed-memory systems, so allowing further scaling to supercomputers. This would, however, require significant reengineering of the skeleton. We also plan to consider more complex real-life applications that conform to the skeleton, to investigate the anomalies that we have observed on e.g. the *zeus* and *phi* systems, and to retry the results for the *power* system once Erlang 18 is available.

References

1. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Pool evolution: a parallel pattern for evolutionary and symbolic computing. IJPP **44**(3), 531–551 (2016)
2. Al-Jaroodi, J., Mohamed, N., Jiang, H., Swanson, D.: Agent-based parallel computing in java proof of concept. Technical Report TR-UNL-CSE-2001-1004, University of Nebraska—Lincoln (2001)
3. Armstrong, J., Virding, S., Williams, M.: Concurrent Programming in Erlang. Prentice-Hall, Englewood Cliffs (1993)

4. Aronis, S., Papaspyrou, N., Roukounaki, K., Sagonas, K., Tsiouris, Y., Venetis, I.E.: A scalability benchmark suite for Erlang/OTP. In: Proceedings of the Erlang Workshop, ACM, Erlang ’12, pp. 33–42 (2012)
5. Back, T.: Selective pressure in evolutionary algorithms: a characterization of selection mechanisms. In: Proceedings of the WCCI, IEEE, vol. 1, pp. 57–62 (1994)
6. Bellifemine, F., Poggi, A., Rimassa, G.: Jade: a FIPA2000 compliant agent development environment. In: Proceedings of the Agents ’01, pp. 216–217 (2001)
7. Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliott, A.: Cost-directed refactoring for parallel Erlang programs. IJPP **42**(4), 564–582 (2014)
8. Byrski, A., Schaefer, R., Smołka, M.: Asymptotic guarantee of success for multi-agent memetic systems. Bull. Polish Acad. Sci. Tech. Sci. **61**(1), 257–278 (2013)
9. Byrski, A., Drezewski, R., Siwik, L., Kisiel-Dorohinicki, M.: Evolutionary multi-agent systems. Knowl. Eng. Rev. **30**, 171–186 (2015)
10. Cahon, S., Melab, N., Talbi, E.: Paradiseo: a framework for the reusable design of parallel and distributed metaheuristics. J. Heuristics **10**(3), 357–380 (2004)
11. Cicirelli, F., Nigro, L.: An agent framework for high performance simulations over multi-core clusters. In: Proceedings of the AsiaSim 2013, Springer, pp. 49–60 (2013)
12. Claessen, K., Sheeran, M., Svensson, B.J.: Expressive array constructs in an embedded GPU kernel programming language. In: Proceedings of the DAMP, pp. 21–30 (2012)
13. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Elsevier **30**, 389–406 (2004)
14. Cosenza, B., Cordasco, G., De Chiara, R., Scarano, V.: Distributed load balancing for parallel agent-based simulations. In: Proceedings of PDP 2011, pp. 62–69 (2011)
15. Digalakis, J., Margaritis, K.: An experimental study of benchmarking functions for evolutionary algorithms. Int. J. Comput. Math. **79**(4), 403–416 (2002)
16. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: a heterogeneous parallel language. In: Proceedings of DAMP ’07, ACM, pp. 37–44 (2007)
17. Fogel, D.B.: What is evolutionary computation? IEEE Spectr. **37**(2), 26, 28–32 (2000)
18. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw. Pract. Exp. **40**(12), 1135–1160 (2010)
19. Gutknecht, O., Ferber, J.: Madkit: a generic multi-agent platform. In: Proceedings of AGENTS ’00, pp. 78–79 (2000)
20. Janjic, V., Brown, C., Hammond, K.: Lapedo: Hybrid skeletons for programming heterogeneous multicore machines in Erlang. In: Proceedings of ParCo 2015, IOS Press (2015)
21. Krzywicki, D., et al.: Massively concurrent agent-based evolutionary computing. J. Comput. Sci. **11**, 153–162 (2015)
22. Lee, H.J., et al.: Implementing domain-specific languages for heterogeneous parallel computing. Micro IEEE **31**(5), 42–53 (2011)
23. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: MASON: a multiagent simulation environment. Simulation **81**(7), 517–527 (2005)
24. Marlow, S., Newton, R., Peyton Jones, S.: A monad for deterministic parallelism. In: Proceedings of Haskell ’11, pp. 71–82 (2011)
25. North, M., Collier, N., Ozik, J., Tatara, E., Macal, C., Bragen, M., Sydelko, P.: Complex adaptive systems modeling with repast simphony. Complex Adapt. Syst. Model. **1**(1), 3 (2013)
26. Piętak, K., Kisiel-Dorohinicki, M.: Transactions on computational collective intelligence X, springer, chap agent-based framework facilitating component-based implementation of distributed computational intelligence systems (2013)
27. Rashkovskii, Y.: Genomu: a concurrency-oriented database. In: Erlang Factory SF 2013 (2013)
28. Reed, R.: Scaling to millions of simultaneous connections. In: Erlang Factory SF (2012)
29. Sagonas, K., Winblad, K.: More scalable ordered set for ETS using adaptation. In: Proceedings of Erlang Workshop, Erlang ’14 (2014)
30. Trinder, P.W., Hammond, K., Loidl, H.W., Peyton Jones, S.L.: Algorithm + strategy = parallelism. J. Funct. Program. **8**(1), 23–60 (1998)
31. Wilson, K.: Migrating a C++ team to using Erlang to deliver a real-time bidding ad system. In: Erlang Factory SF (2012)
32. Zheng, S., Long, X., Yang, J.: Using many-core coprocessor to boost up Erlang VM. In: Proceedings of Erlang Workshop, ACM, pp. 3–14 (2013)