Manuscript Number: JOCS-D-15-00237

Title: Highly Scalable Erlang Framework for Agent-based Metaheuristic Computing

Article Type: SI: Complex Metaheuristics

Keywords: Metaheuristic Computing; Concurrent programming; Scalability; Erlang

Corresponding Author: Dr. Aleksander Byrski, Ph.D.

Corresponding Author's Institution: AGH University of Science and Technology

First Author: Wojciech Turek, PhD.

Order of Authors: Wojciech Turek, PhD.; Jan Stypka, BSc; Daniel Krzywicki, MSc; Piotr Anielski, MSc; Kamil Pietak, MSc; Aleksander Byrski, Ph.D.; Marek Kisiel-Dorohinicki, PhD

Abstract: Difficult search and optimization problems, usually solved by metaheuristics, are very often implemented in concurrent and parallel environment, as many metaheuristics (e.g. population- or agent-based) are inherently easy to parallelize. Therefore search for easy-to-use, robust and efficient frameworks dedicated for such computing methods, especially during the era of ubiquitous many and multi-core system is very desirable.
Indeed, the development of multi-core architectures is incredibly fast and multicore CPUs can be found nowadays not only in supercomputers, but also in ordinary laptops or even phones. Efficient use of multicore architectures requires applying suitable languages and technologies, like Erlang. Its concurrency model, based on lightweight processes and asynchronous message-passing, seems very well suited for running massively concurrent code on many cores.
Most of existing Erlang industrial applications are deployed on computers with up to 24 CPU cores, and there are hardly any reports on using Erlang on architectures exceeding 32 physical cores. In this paper we present our experiences with developing a concurrent Erlang-based computing platform, scaling a computationally-intensive and memory-intensive applications up to 64 cores, using as examples global optimization and urban traffic planning problems.

Suggested Reviewers: Vladimir Janjic PhD
University of St Andrews, UK
vj32@st-andrews.ac.uk
Expert in parallel programming

Tamas Kozsik PhD
Eotvos Lorand University, Hungary
Tamas.Kozsik@elte.hu
Expert in Erlang programming

Grzegorz M Wojcik PhD
Maria Curie-Sklodowska University, Poland
gmwojcik@umcs.pl
Expert in naturally-inspired computing

Opposed Reviewers:

Cover Letter

Dear Editors

Hereby I submit an original, not previously published paper
"Highly Scalable Erlang Framework for Agent-based Metaheuristic Computing".
The paper is authored solely by: W. Turek, J. Stypka, D. Krzywicki,
P. Anielski, K. Pietak, A. Byrski and M. Kisiel-Dorohinicki.

I kindly ask you for considering this paper for publication in
Special Issue of JOCS on Complex Metaheuristics, guest-edited
by prof. Carlos Cotta and prof. Robert Schaefer.

Best regards
Aleksander Byrski

***Biographies (Text)**

Wojciech Turek, PhD, obtained his PhD in 2010 at 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.

Jan Stypka is a MSc student at AGH University of Science and Technology in Cracow, he is interested in parallel and distributed programming and social networks.

Daniel Krzywicki} is a PhD student at AGH University of Science and Technology in Cracow, he is interested
in agent-based computing and parallel programming in functional languages.

Piotr Anielski obtained MSc in 2013 at AGH University of Science and Technology in Cracow, he is interested
in parallel and distributed programming mostly in functional languages.

Kamil Pietak obtained MSc at AGH University of Science and Technology in Cracow,
he is interested in agent-based frameworks, software engineering, DSLs and component-based systems.

Aleksander Byrski obtained his Ph.D. in 2007 and DSc (habilitation) in 2013 at AGH University of Science and Technology in Cracow.
He works as an 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.

Marek Kisiel-Dorohinicki obtained his Ph.D. in 2001 and DSc (habilitation) in 2013 at AGH University of Science and Technology in Cracow.
He works as an assistant professor at the Department of Computer Science of AGH-UST.
His research focuses on intelligent software systems, particularly utilising agent technology and evolutionary algorithms,
but also other soft computing techniqes.

***Biographies (Photograph)**
Click here to download high resolution image

***Biographies (Photograph)**

Click here to download high resolution image

***Biographies (Photograph)**
Click here to download high resolution image

***Biographies (Photograph)**
Click here to download high resolution image

***Biographies (Photograph)**
Click here to download high resolution image

***Biographies (Photograph)**
Click here to download high resolution image

***Biographies (Photograph)**
Click here to download high resolution image

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Journal of Computational Science 00 (2015) 1–19
Journal Logo

Highly Scalable Erlang Framework for Agent-based Metaheuristic Computing

Wojciech Turek, Jan Stypka, Daniel Krzywicki, Piotr Anielski, Kamil Pietak, Aleksander Byrski, Marek Kisiel-Dorohinicki

AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications
Al. Mickiewicza 30, 30-059 Krakow, Poland

wojciech.turek@agh.edu.pl, janstypka@gmail.com, krzywic@agh.edu.pl, pr.anielski@gmail.com, {kpietak,olekb,doroh}@agh.edu.pl

Abstract

Difficult search and optimization problems, usually solved by metaheuristics, are very often implemented in concurrent and parallel environment, as many metaheuristics (e.g. population- or agent-based) are inherently easy to parallelize. Therefore search for easy-to-use, robust and efficient frameworks dedicated for such computing methods, especially during the era of ubiquitous many and multi-core system is very desirable. Indeed, the development of multi-core architectures is incredibly fast and multicore CPUs can be found nowadays not only in supercomputers, but also in ordinary laptops or even phones. Efficient use of multicore architectures requires applying suitable languages and technologies, like Erlang. Its concurrency model, based on lightweight processes and asynchronous message-passing, seems very well suited for running massively concurrent code on many cores. Most of existing Erlang industrial applications are deployed on computers with up to 24 CPU cores, and there are hardly any reports on using Erlang on architectures exceeding 32 physical cores. In this paper we present our experiences with developing a concurrent Erlang-based computing platform, scaling a computationally-intensive and memory-intensive applications up to 64 cores, using as examples global optimization and urban traffic planning problems.

Keywords: Metaheuristic Computing, Concurrent programming, Scalability, Erlang

1. Introduction

Tackling difficult search problems calls for applying unconventional methods. This necessity is imposed by having little or no knowledge of the intrinsic features of the problem, topology of search space etc. In such cases, approximate techniques, like metaheuristics become the methods of last resort. Having a plethora of metaheuristics to choose from, those population-based (as opposed to single solution oriented) seem to be the best choice, both at algorithmic and implementation level, and as they process more than one solution at a time, they can evade local extrema easier than single-solution approaches. Moreover, it is easy to implement them efficiently using ubiquitous parallel systems, such as multi-core processors, graphical processing units, clusters and grids.

Evolutionary processes are by nature decentralised and therefore evolutionary processes in a multi-agent system at a population level may be easily introduced. It 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). This idea came into fruition by Cetnarowicz in 1996 [1] as Evolutionary Multi-agent System, and since that time implemented (cf., e.g., [2, 3]) a number of times, analysed [4] and extended [5, 6]. It is to note, that EMAS turned out to be an efficient paradigm for solving different optimization problems [7, 8].

The development of multi-core architectures over the last ten years is amazingly fast. Mainstream computer components manufacturers compete with smaller companies and startups in designing more and more sophisticated architectures, which include tens (soon hundreds) of cores in a single processor. Architectures like Intel Xeon Phi [9] or Adapteva Epiphany [10] are typically available as dedicated accelerators, however, general-purpose many-core CPUs, like the 100-core EZchip TILE-MX [11] will soon become standard equipment of computing stations.

This rapid development poses significant challenges for the software industry which seems unprepared for using this computational power. The growing number of independent cores available for a single process makes the most popular software development technologies inefficient. Being based on imperative programming paradigm and shared memory concurrency model, the technologies are unsuitable for handling hundreds of simultaneously running tasks.

In this paper we present a new approach to the problem of building a framework for metaheuristic computing. By using Erlang programming language and its virtual machine, which natively supports lightweight processes and message-passing concurrency, we managed to provide linear scalability of computing performance on a many-core architecture. We show that the scalability of certain internal mechanisms of the Erlang VM is limited (Erlang VM is written in C, an imperative language). We present several methods of diagnosing the reasons of particular problems and solutions to improve the scalability of Erlang, in order to achieve proper scalability on truly large amounts of computing cores.

The created framework has been tested on two different types of computations. Firstly we used a typical benchmark function with many local minima, which required time-consuming evaluation without complex operation on memory. In order to further evaluate features of the developed solution, a real life optimization problem has been considered. The problem of micro-scale urban traffic planning was solved using a novel, multi-variant planning approach, which continuously prepares various solutions to the most probable situations on the managed crossroad. This problem required performing costly computations together with memory-intensive operations.

After the introduction, we present our parallel computing model (an evolutionary multi-agent system, eMAS), as a universal optimization algorithm, along with the description of its parallel implementation using the functional approach. Then, we present the architecture of our Erlang computing framework and the details of four different implementations of eMAS. Next, we show how to scale the Erlang VM up to 64 cores, by reporting the problems we encountered and the solutions we found, along with experimental results illustrating consecutive steps. Finally, we describe a real-life problem of traffic management planning, showing its implementation, results and conclusions.

2. Agent-oriented frameworks and computing

Agent-based software environments use agents as basic units of software abstraction. They focus on inter-agent relation and intra-agent intelligence, provide facilities for the discovery of agents, communication, life-cycle management etc. The FIPA standard allows to create such open, interoperable multi-agent systems, where fully-fledged autonomous software agents can express their needs or perceive the environment (and other agents) using specific languages, ontologies, etc. The most established solutions of this kind include JADE [12] and JADEX [13]. However, the approach of open systems with heavy agents is not applicable for computational systems – granularity of agents is similar to services in SOA.

In case of simulations or computations, where the introduction of agents facilitates the modelling of complex phenomena, such as natural or social ones, agents constitute building blocks of the model. This approach is utilized in frameworks such as Mason, RePast or MadKit.

The first one, **MASON** [14], is a single-process discrete-event simulation core and visualization library written in Java developed at George Mason University. It is supposed to support efficiently up to a million agents without visualization facilities. The multi-layer architecture brings complete independence of the simulation logic from visualisation tools. There are none ready-to-use distributed computing facilities, however it can be integrated into larger existing libraries. The programming model of MASON follows the basic principles of object-oriented design. Agents are lightweight entities represented as Java objects with step method. They are added to a scheduler and their step method is called during the simulation. Agents API is flexible and allows to model various computation models, but there is no built-in metaheuristics.

RePast [15] is a widely used agent-based modelling and simulation tool. It has multiple implementations in several languages (e.g RePast Simphony in Java and RePast HPC written in C++) and built-in adaptive features such as genetic algorithms and regression. The framework uses fully concurrent discrete event scheduling. Dynamic access to the models in the runtime (introspection) is possible using a graphical user interface. The RePast distribution has a large footprint: included in the package areneural networks, genetic algorithms, social network modeling, dynamic systems model-ing, logging, GIS, and graphs and charts [14].

MadKit is a modular and scalable multi-agent platform written in Java, aimed at modelling different agent organisations, groups and roles in artificial societies. It is built based on a so called Agent/Group/Role organisational model, using a plugin-based architecture. The architecture of MadKit is based on micro-kernels which provide only the basic facilities: local messaging, management of groups and roles, launching and killing of agents. Other features (remote messages, visualisation, monitoring and control of agents) are performed by agents. Both thread based and scheduled agents may be developed.

All of the above platform provide wide range of facilities supporting agent-based computations and simulations. They all are built using imperative languages such as Java or C++. Agents are represented as simple objects sequentially executed by a scheduler (so called steppable agents) or as heavy agents usually executed as separated threads.

The community of language developers seem to agree that a paradigm shift is needed in order to adopt to the challenges of modern hardware [16]. Efficient development of software for many-core architectures will require high level functional languages with immutable variables, no shared state and message-passing concurrency. Some of these relatively old concepts are being adopted in new and existing languages, like C++, Java or Scala. These are also the most basic assumptions of Erlang programming language.

Erlang [17] is a high level functional language which has proven to be an efficient tool for building large-scale systems for multi-core processors. The concurrency model based on lightweight processes seems very well suited for running massively concurrent code on many cores or processors. Therefore, it might seem that Erlang is a good programming choice for parallel computing on many-core architectures. Finding out that it is not that simple cost us a lot of research effort, which we summarize in this paper.

Most of Erlang industrial applications are deployed on computers with up to 24 CPU cores. Scalability of solutions is provided by using clusters of computers – running Erlang in distributed configuration. Actually, the OTP team developing the Erlang Virtual Machine does not test the implementation on bigger architectures[1].

There are few reports on using a single Erlang Virtual Machine on architectures exceeding 32 physical cores. In [18] the authors present a test suite for measuring different aspects of Erlang applications performance. The exemplary test run on 64 core machine show 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 [19]. Promising results of using Erlang on a Intel Xeon Phi coprocessor have been shown in [20]. Basic benchmarks show good scalability up to 60 cores, which is the number of physical cores of the coprocessor. However, there are hardly any reports on scaling complex, computationally intensive Erlang applications on many-core architectures.

The computing framework presented in this paper has a counterpart implemented in Scala[2] [21], being a subject of research on the same concurrent implementation style yet using other technology. The comparison of these platform became the main subject of other publications of our research team.

3. Computing models for Agent-based metaheuristics

Various models of parallel implementations of evolutionary algorithms have already been proposed [22]. The standard approach (sometimes called a *global parallelisation*) consists in distributing selected steps of the sequential algorithm among several processing units. *Decomposition* approaches are based on defining different complex models such as *coarse-grained* and *fine-grained* parallel evolutionary algorithms. There are also methods which use some combination of the models described above (*hybrid* parallel evolutionary algorithms).

Agents play an important role in the integration of artificial intelligence subdisciplines, which is often related to a hybrid design of modern intelligent systems [23]. In some similar applications reported in the literature (see,

[1]Based on a discussion with an OTP member at the Erlang User Conference in Stockholm, 2014

[2]http://github.com/ParaPhraseAGH/scala-mas

e.g. [24, 25] for a review), an evolutionary algorithm is used by an agent to facilitate the execution of some of its tasks, often connected with learning or reasoning, or to support coordination of some group (team) activity. In other approaches, agents constitute a management infrastructure for a distributed realisation of an evolutionary algorithm [26]. A quite similar approach is proposed by Liu et al. [27, 28] where agents are situated on the lattice, derivatives of their fitness are encoded in a form of energy function and dedicated operators for reproduction and removal of the individuals are introduced (mimicking cellular evolutionary algorithms [29]).

In this section, staring from an EMAS [1] being an example of general-purpose agent-based metaheuristic, concurrent, scalable way of implementation and relevant Erlang-based software platform are presented. This platform may be updated and adapted to other agent-oriented metaheuristics, however agency is crucial for the next-to-come applications of the framework, as lightweight agent execution became a starting point and the inspiration for the whole research presented in this paper.

3.1. Evolutionary Multi-Agent System

Evolutionary multi-agent systems are a hybrid meta-heuristic which combines multiagent systems with evolutionary algorithms. The idea consists in evolving a population of agents to improve their ability to solve a particular optimization problem [30, 4].

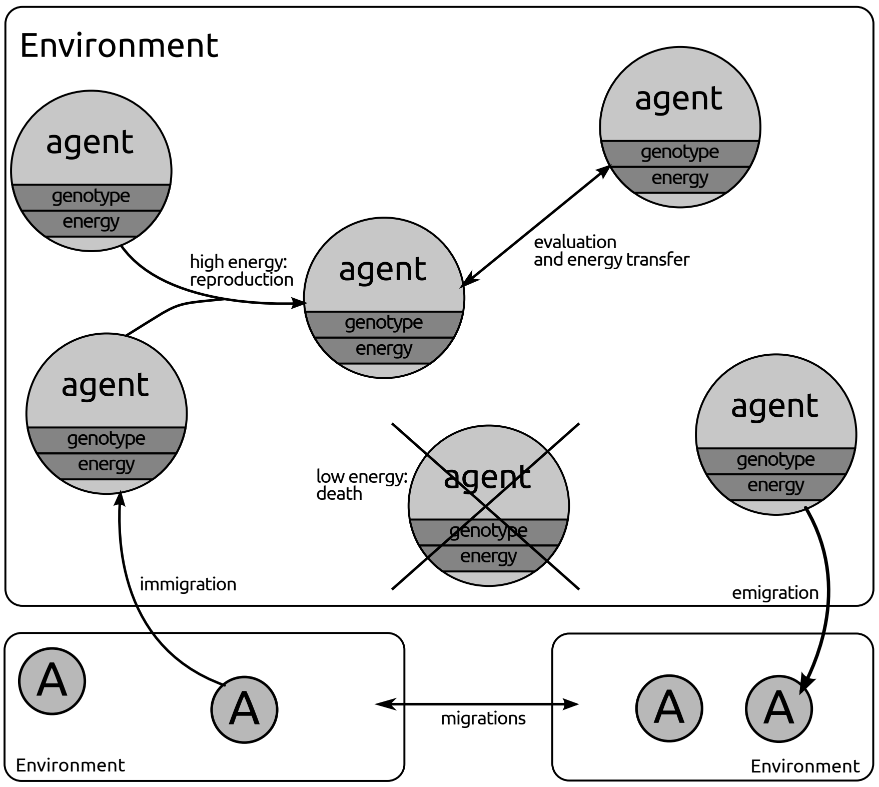

Figure 1. EMAS structure and principle of work

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, selective pressure needs to be decentralized, in contrast with traditional evolutionary algorithms. 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 with a real-valued vector representing a potential solution to the optimisation problem, along with the corresponding fitness. 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. The system is stable as the total energy remains constant, but the number of agents may vary and adapt to the difficulty of the problem (see Fig. 1).

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 were formally proven by constructing a detailed Markov-chain based model and proving its ergodicity [4], showing EMAS as a general optimization tool.

3.2. Interaction and Execution Models for Agents

In agent-oriented computing systems, agent interactions are one of the crucial aspects of their work. It is easy to predict that parallelising them can significantly increase the throughput of the system. However, this comes at the cost of increased communication and synchronisation. Therefore, an important issue is to choose the appropriate granularity of the entities in the computation.

As agents are defined as autonomous and independent beings, it seems natural to look for further concurrency within a single environment. The question is where to put the boundaries of concurrent execution, as it has consequences on both performance and ease of programming. This section discusses the most common models of execution and interaction in existing agent software [31].

Heavyweight Agents. In this model every agent is associated with a thread and communicates through message passing. Some agents may passively wait for incoming messages and react to them. Other agents may actively initiate interactions with other agents. It is difficult to achieve a coordinated life cycle among such agents, since the corresponding threads may be arbitrary interleaved. Therefore, some kind of synchronisation between agents still needs to be introduced, usually in terms of a specific communication protocol.

In order to interact with each other, agents need to locate other agents willing to perform the same actions. For example, in an evolutionary multi-agent system, an agent with enough resources to reproduce needs to find another one which also has enough resources. In order to do that, it could ask all other agents in the population. However, such a solution is obviously inefficient, because of the intensity and redundancy of the required communication.

A better approach, introduced in [31], is to use a mediating entity, which we call a *meeting arena*. Every time an agent wants to perform an action, it chooses an appropriate arena to meet with other similar agents (see Listing. 1). The arena is then able to partition its members in groups of some given arity and mediate the meeting itself (see Listing 2 and Fig. 2).

```
behaviour(Agent) when Agent#agent.energy == 0 -> death
behaviour(Agent) when Agent#agent.energy > 10 -> reproduction
behaviour(Agent) -> fight
```

Listing 1. In every step, agents choose an arena based on their current state

```
meeting({death, Agents}) -> []
meeting({reproduction, Agents}) ->
  lists:flatmap(
    fun doReproduce/1,
    inGroupsOf(2, Agents));
meeting({fight, Agents}) ->
  lists:flatmap(
    fun doFight/1,
    inGroupsOf(2, Agents));
```

Listing 2. Arenas process partitions of the population and trigger agent meetings.

The usage of meeting arenas should bring many benefits, not only in terms of efficiency, as the algorithm itself can be structured more clearly. Agents only need to be given a set of rules, in order to choose an arena on the basis of their state. The actual protocol of agents interactions can then be defined at the level of the appropriate arena.

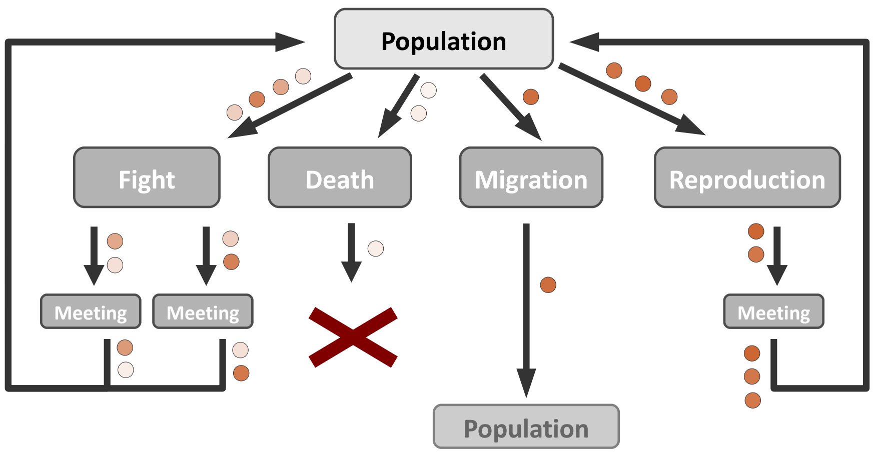

Figure 2. Meeting arenas allow to group similar agents and coordinate meetings between them.

Assigning a thread to each agent may feel very natural. In practice, however, the number of agents is often much higher than the number of cores. Performance may then be seriously hindered by frequent context switches, although this overhead may be reduced by sharing a pool of threads among agents. However, this model still involves intensive communication and costly processor cache synchronisation. In consequence, the trade-off for such concurrency may be higher than expected.

Lightweight Agents. An opposite approach is to consider agents as parts of the model, but not parts of the implementation. As such, they are simply represented as data structures and processed like in a discrete event simulation.

The execution of an individual agent has to be divided into smaller parts which can be interleaved. These parts, which we will call *actions*, could for example consist of moving to different location or meeting a neighbour. Given its current state, every agent decides which action to perform next. Then the agents are grouped by the selected action and the actions are performed on pairs or individual agents. Finally the new population is shuffled to change the order of agents before next iteration of the algorithm (listing 3).

```
update(Agents) ->
  Tagged = [{behaviour(A), A} || A <- Agents],
  Grouped = group_by_behaviour(Tagged),
  NewAgents = lists:flatmap(fun meeting/1, Grouped)
  shuffle(NewAgents)
```
Listing 3. The behaviour and meeting functions are applied to derive the next agent population.

The performance of such a model will usually be higher than in the previous one, as more consistent memory access patterns result in more efficient processor usage. Even though the explicit parallelism is reduced, throughput can be improved, because frequent agent interactions no longer need to be synchronised between threads.

Moreover, independent actions can still be executed in parallel by the executor service. This is consistent with the meeting arena concept described above, as actions on common subsets of agents may be grouped together and considered as a single meeting.

4. Framework Architecture and EMAS Implementation

The platform is thus divided into two layers: the multi-agent application and the execution environment (Fig. 3).

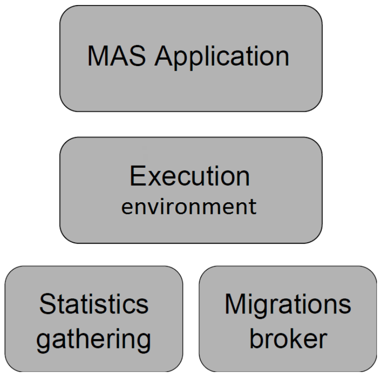

Figure 3. The architecture of the platform.

The application layer abstracts from the execution and focus on the interactions. It defines

- the types of agents
- their possible states and actions
- the behaviour function mapping states to actions
- the meetings function updating agents subpopulation for a given action

The execution environment implements the logic of computing of these functions (realizing them in the form of one of the described below execution models) and tying their results together to run the simulations. Several versions of execution models are described below, but they all expose the same API, accepting the types and functions defined at the application layer.

This decoupling allows to design the multi-agent algorithm separately and later choose and tune the execution model to best fit a given problem or hardware configuration.

A broker module is responsible for handling agent migrations between environments, independently of their underlying execution environment. Although this is outside the scope of this paper, the same mechanism also allows the application to handling cross-node migrations in a distributed setting.

The last module is responsible for gathering statistics and metrics from the running application. Several mechanisms are available for different levels of concurrency. In particular, a special support is provided for statistics that are monotonic, such as the number of agents meetings or the best fitness found so far. Such statistics can be computed at any time and recorded asynchronously. They can be periodically *flattened* to yield the value of the statistic at a given time, with minimal synchronisation between sources.

As described in 3.2, choosing an appropriate execution model for agent interactions is crucial to achieve efficient parallelism in multi-agent systems. Such execution models may differ in terms of granularity and other characteristics, in particular their efficiency for a given application. Decoupling agent behaviour and meetings from the interaction mechanism [31] allows to swap different execution models and compare their performance.

We designed three different implementations of agent interactions: hybrid (coarse-grained), concurrent (fine-grained) and SKEL-based (also coarse-grained, based on the SKEL [32] library[3]). For comparison, we also implemented a fully sequential version.

Sequential. The sequential version is very simple: it repeatedly applies the behaviour and meeting functions (see Sec. 3.2) to update the population of agents (Listing 3). Additionally, several such agent populations are kept and in every step agents may migrate between them with low probability. All the populations are then wrapped in a recursive loop within a single Erlang process until some stop criterion is met.

[3]SKEL is one of the products of EC FP7 ParaPhrase Project, contract no. 288570, http://paraphrase-ict.eu/

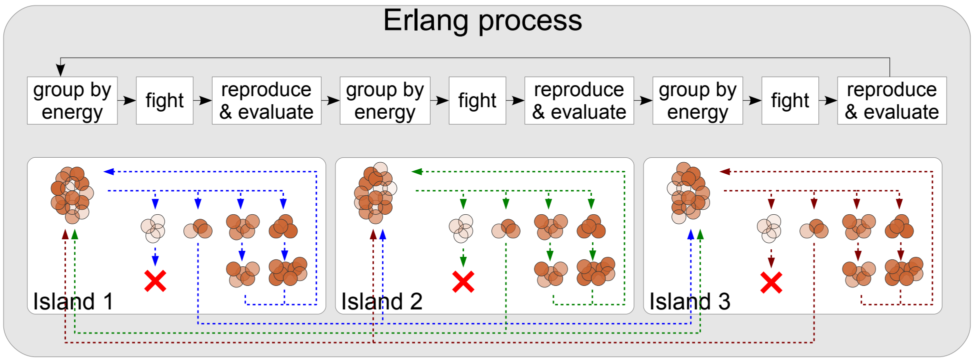

Figure 4. Outline of the sequential version of EMAS.

Figure 4 presents the outline of the sequential is implementation. Single Erlang process iterates over the collection of islands. Within each island solutions are grouped by energy value into three groups: the agents to remove, fight and reproduce. Additionally, a group of agents for migration is selected with low probability from the last two groups. In following steps the process performs migration (moves the solutions to another agents collection) and meetings between agents. Operation of fighting results in equinumerous set of agents, reproduction causes growth in the number of agents. New solutions are evaluated during this process, which is the most computationally-expensive operation of the whole process.

Hybrid. The hybrid version is the most straightforward way to parallelize the basic, sequential version. Every agent population is contained in a separate Erlang process and runs in an independent loop. The outline of this implementation is presented in figure 5.

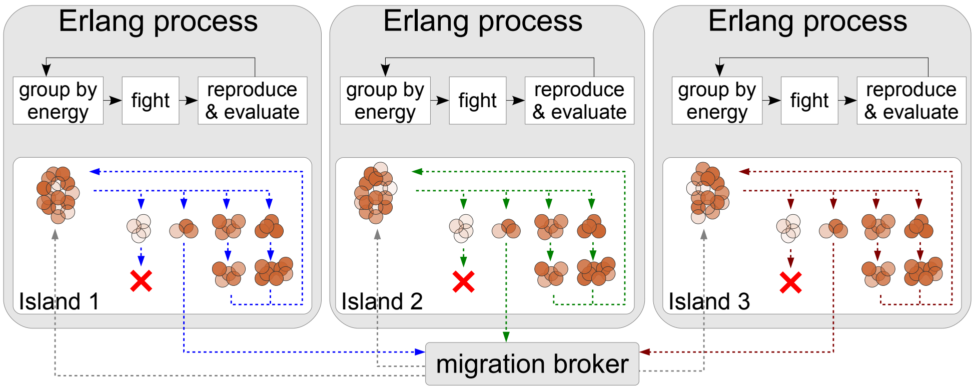

Figure 5. Outline of the hybrid version of EMAS.

Agent migration is realized through message passing. A single, separate process acts as a migration broker and is responsible for forwarding migrating agents according to some topology. In practice, the migration probability is very low (typically a few migrations every second), so this single process does not become a bottleneck but simplifies communication.

Skeletons. The SKEL-based implementation is the result of refactoring the sequential version by introducing skeletons from the SKEL library. These skeletons are used to emulate loops and parallel patterns in the code. The outline of this implementation is presented in figure 6.

Agent populations are parallelized with the map skeleton and the feedback pattern is used to loop each population independently. While the rest of the logic is implemented with skeletons, asynchronous migration had to be written separately. Agents are exchanged between islands through an ETS table which is fast and threadsafe.

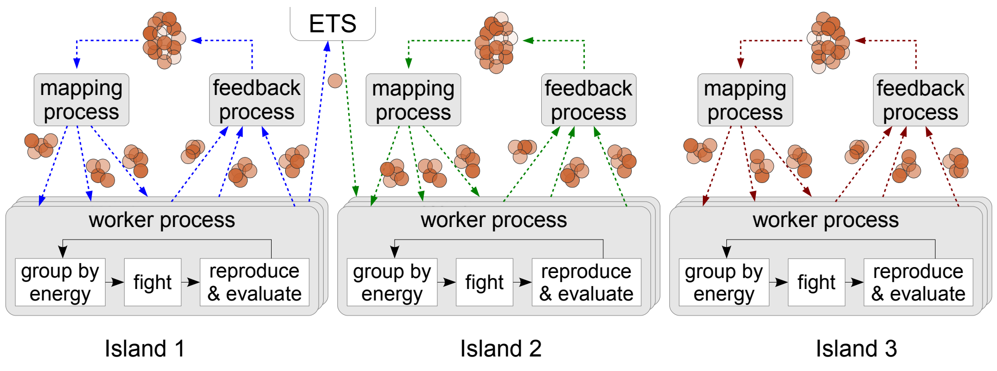

Figure 6. Outline of the SKEL version of EMAS.

The codebase of this implementation is smaller while the final performance is almost as good as in the hybrid model. Also, the structure of the parallel workflow can be easily changed. On the other hand, this structure highly influences the performance, so we had to experiment with several versions until good performance was achieved.

Concurrent. The concurrent version represents an approach where every agent runs in a separate autonomous Erlang process. As they lack global knowledge, agents only interact through the mediating meeting arenas. In these model the arenas become distinguished entities. They are implemented as Erlang processes (gen_servers) constant for each island. There is a separate arena responsible for each kind of interaction: fight, reproduction, death and migration. The outline of this implementation is presented in figure 7.

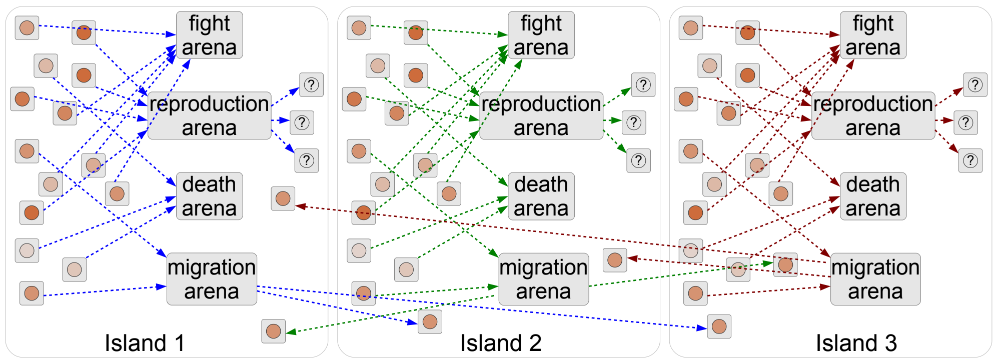

Figure 7. Outline of the concurrent version of EMAS.

When agents decide to carry out a certain action e.g. reproduce, they send a message to an appropriate arena. The arena pairs such incoming agents, calculates the interaction and sends back its results. This approach allows to keep very lightweight agents and removes all synchronisation barriers from the algorithm, at the cost of more expensive communication.

Every agent population has a separate set of arenas. Migrating an agent is as simple as changing the PIDs of the arenas it communicates with.

5. Scalability of the platform applied for benchmark continuous optimization

In this section the scalability issues encountered during optimization of the framework applied for continuous optimization (Rastrigin function) are discussed.

All the versions of the algorithm described in the previous section have been carefully tested to evaluate their performance, scalability and compare with each other. To our surprise, initial implementations off all of them did not scale well for large numbers of CPU cores.

The performance metric recorded in our experiments is the number of agent reproductions happening every second. Preliminary work indicated that this amount is proportional to the amount of all other events in the system and thus a good indication of the system's overall throughput. However, concrete throughput values are problem dependent and should not be compared across use-cases.

We ran our experiments on the ZEUS supercomputer provided by the Pl-Grid [4] infrastructure at the ACC Cyfronet AGH [5]. We used nodes with 4 AMD Opteron 6276 processors each (64 cores per node) and a total of 256 GB of memory per node.

For every experiment, we started the computations and let the system reach a steady state. Then, we recorded throughput during several minutes. This procedure was repeated 30 times to account for operating system variability or dependence on random number generator seed. Finally, these throughput were averaged over time and experiments for a given configuration.

Optimizing the Rastrigin benchmark function [33] is a computationally intensive task. By varying the dimension of the problem, we can choose appropriate computation to communication ratios to identify scalability problems related with communication.

Our initial implementations were not very scalable, as throughput increased only up to 8 cores. Above this threshold, only the hybrid version further improved (Fig. 8).

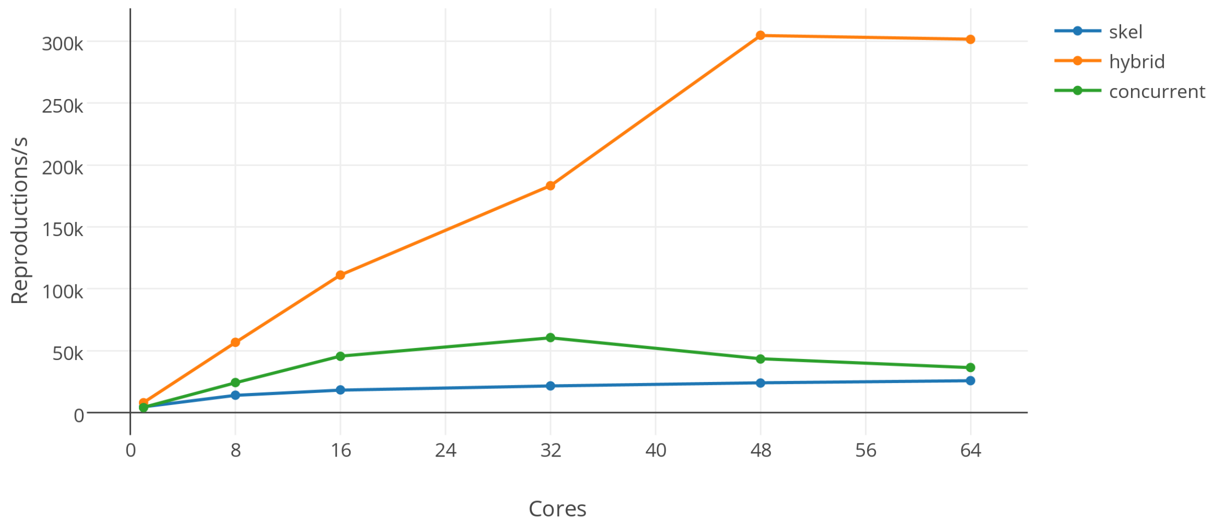

Figure 8. Throughput of initial implementations. Only the hybrid version scales to some extent.

It was not easy to profile the code and look for bottlenecks, as most of common Erlang profiling tools are not designed for HPC programs. We managed to monitor the schedulers' load which was constantly 100% and employed tools such as *eprof*, *fprof* or *percept*, however those approaches did not yield any satisfying results.

We also used the *lcnt*[6] tool. This library provides functions to monitor lock contention in the virtual machine and provides valuable information concerning the number of collisions and time spent on each lock.

After running our application with lock monitoring enabled, we discovered that the concurrent model suffers from extensive usage of the *make_ref/0* function. In Erlang this function is responsible for generating a globally unique identifier and has to be synchronized between schedulers. It is also used by the *gen_server:call/2* function, which was used extensively in our program. In the concurrent model, arenas have been implemented as *gen_server* behaviours. Therefore, every time agent-arena communication occurred, the *make_ref/0* function had to be called, which resulted in a big performance loss. Changing communication to use the *gen_server:cast/2* function proved essential to good scalability.

[4]http://www.plgrid.pl/en
[5]http://www.cyfronet.krakow.pl/en/
[6]http://www.erlang.org/doc/man/lcnt.html

Another significant change was the introduction of the *exometer*[7] library as a global logging system. Thanks to its built-in counters, we managed to lighten the arenas. However, statistics gathering was very difficult due to the distributed and decentralized nature of our algorithm. After several approaches, we implemented custom NIF functions for that purpose which performed very well.

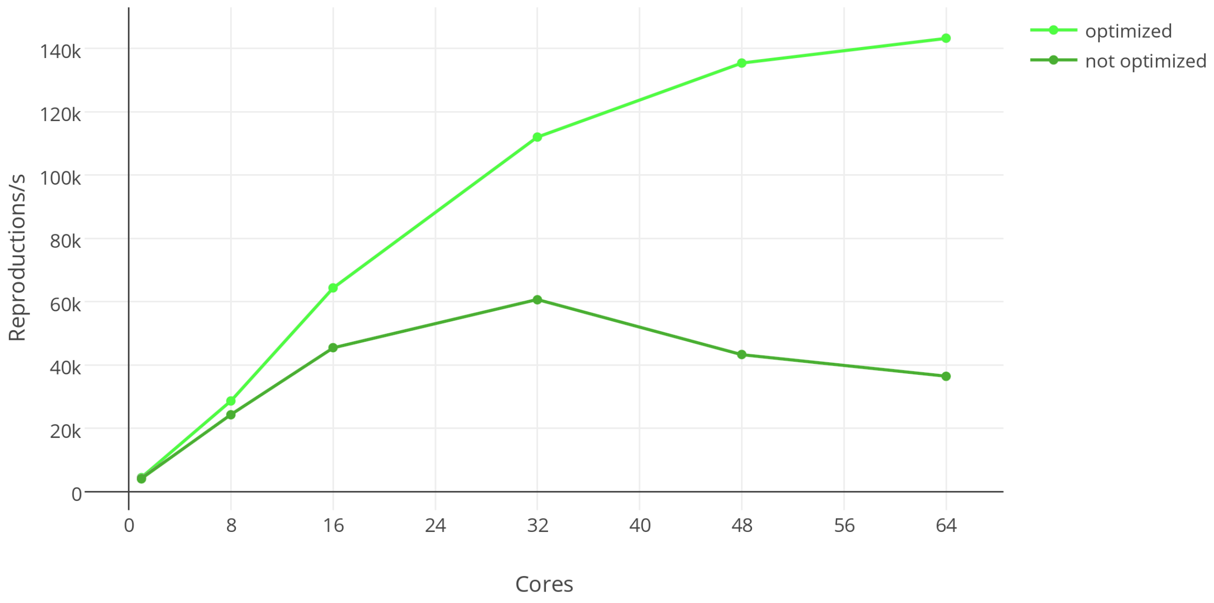

Figure 9. Throughput of the concurrent model after optimizing both communication and logging.

The performance improvement of the concurrent version after changes in communication and logging is shown in Fig. 9.

However, no locks were observed in the skel model. Therefore, we experimented with the skeleton structure in order to increase performance. The main challenge was related to agent migrations - it was impossible to implement them through message passing, because skel processes are transparent for the programmer and it is difficult to get their PIDs. Our first approach consisted of adding a synchronisation barrier between islands and performing synchronized migrations after each algorithm iteration (Listing 4). This simple solution proved to be not scalable, as the synchronous migration become a bottleneck and slowed down the computation significantly.

```
Map = {map, [MainWorkflowFun], Workers},
Feedback = {feedback, [Map, MigrationFun], StopConditionFun},
skel:do([Feedback], [Population]).
```

Listing 4. First approach to skel-based program implementation.

In order to solve this problem, we implemented migration separately using ETS tables and removed the synchronization point. In this approach each island iterated in its own loop and asynchronous communication was emulated through writing and reading the ETS tables (Listing 5). The performance improvement of the skel version after reordering skeletons is shown in Fig. 10.

Skel patterns and workflows look very simple for the user, however underneath they involve a lot of message passing through many different processes. In Erlang every outgoing message has to be copied in memory, therefore sending large messages may be computationally expensive. However, this is not the case for large instances (more

[7]https://github.com/Feuerlabs/exometer

```
Feedback = {feedback, [MainWorkflowFun], StopConditionFun},
Map = {map, [Feedback], Workers},
skel:do([Map], [Population]).
```

Listing 5. New skel-based implementation logic.

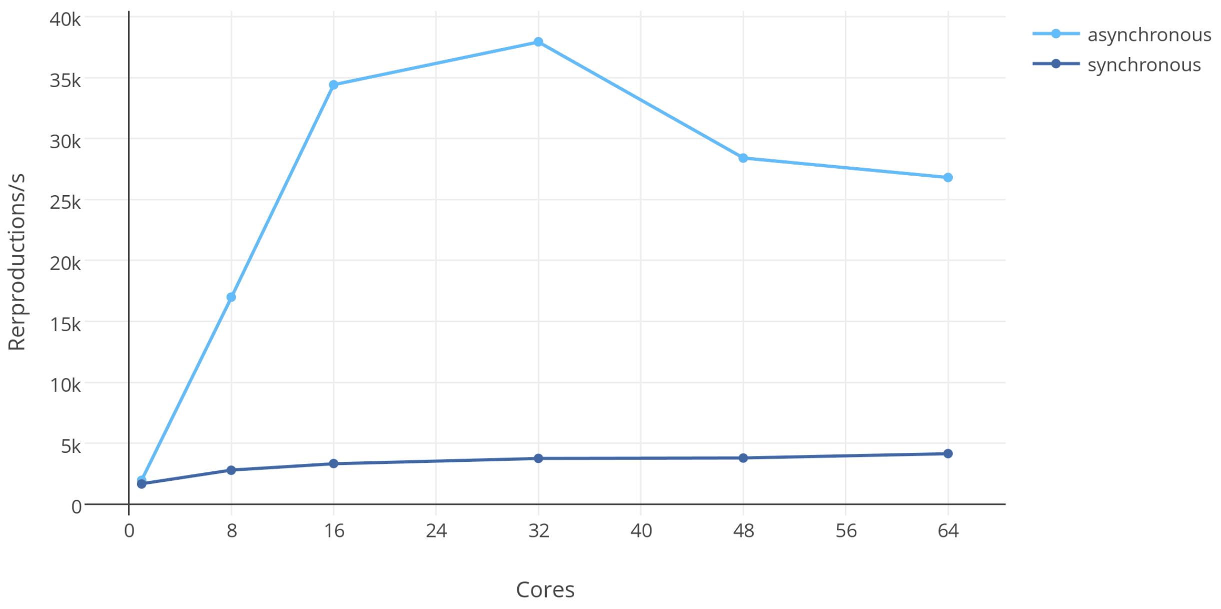

Figure 10. Throughput of the skel model depending on skeleton structure and agent migration approach: synchronisation barrier (synchronous) or ETS tables (asynchronous)

than 64 bytes) of the Erlang binary data type. Large Erlang binaries are held in a separate space in memory and sent by reference instead of by value. In order to further improve skel version's performance, we moved from a list representation of agents to binary types. We experienced a significant performance and scalability boost, as most of communicational overhead vanished (Fig. 11).

Fig. 12 shows the final scalability of all three implementations. Achieved scalability makes it possible to efficiently utilize a 64 core CPU. This results prove that it is possible to successfully use Erlang for computations on many-core architectures, however getting linear scalability of a particular algorithm is not straightforward.

6. Urban Traffic Management

In order to confirm the scalability features of the developed framework, a more complex, real-life problem has been considered. The computing framework has been applied for optimizing traffic on a simulated urban crossroad. This use case was selected as being a much more practical one than popular and acclaimed, yet quite artificial benchmark functions. Moreover, this problem employs a data-intensive fitness function, again, contrary to a very simple-to-compute Rastrigin function (or similar benchmarks).

Traffic affects all people living in crowded cities, but despite many years of research in this area, most of crossroads are still controlled by simple sequence-based traffic lights. The nature of the problem makes it very hard to be solved both safely and efficiently. Efficient coordination of motion for tens of cars requires complex computations, while high dynamics puts very strong constraints on computations time and robustness. Moreover, the plans created are never executed accurately, therefore some uncertainty must be explicitly accounted for in the planning algorithm. These factors make the classic, domain independent planning methods [23] unsuitable.

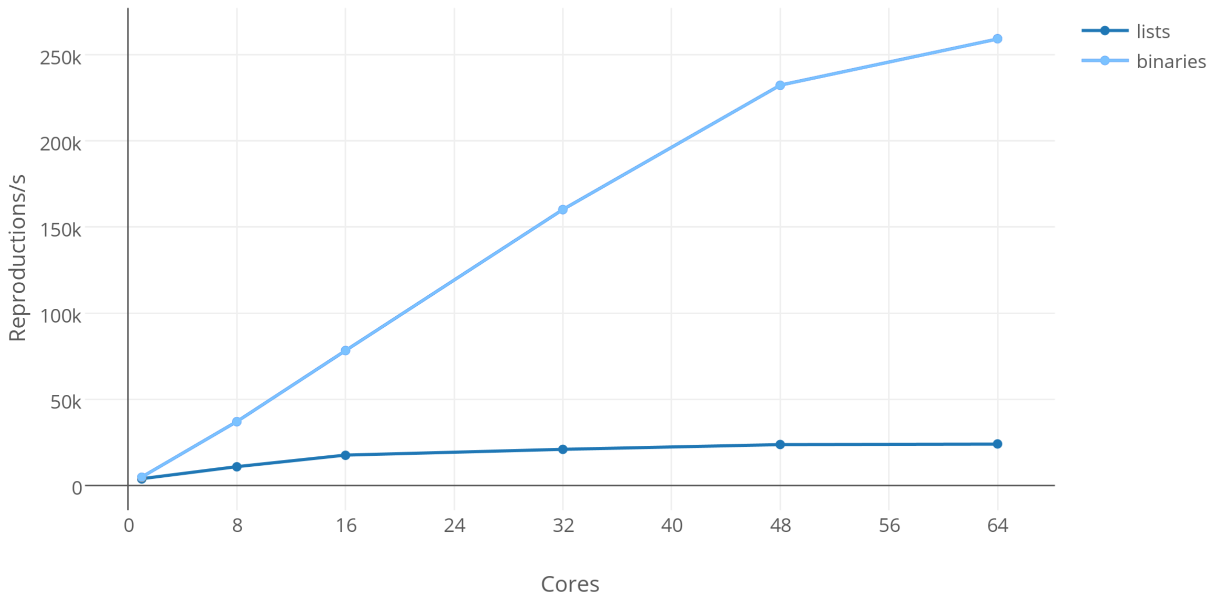

Figure 11. Performance boost of the skel version after changing agent representation from list (sent by value) to binaries (sent by reference).

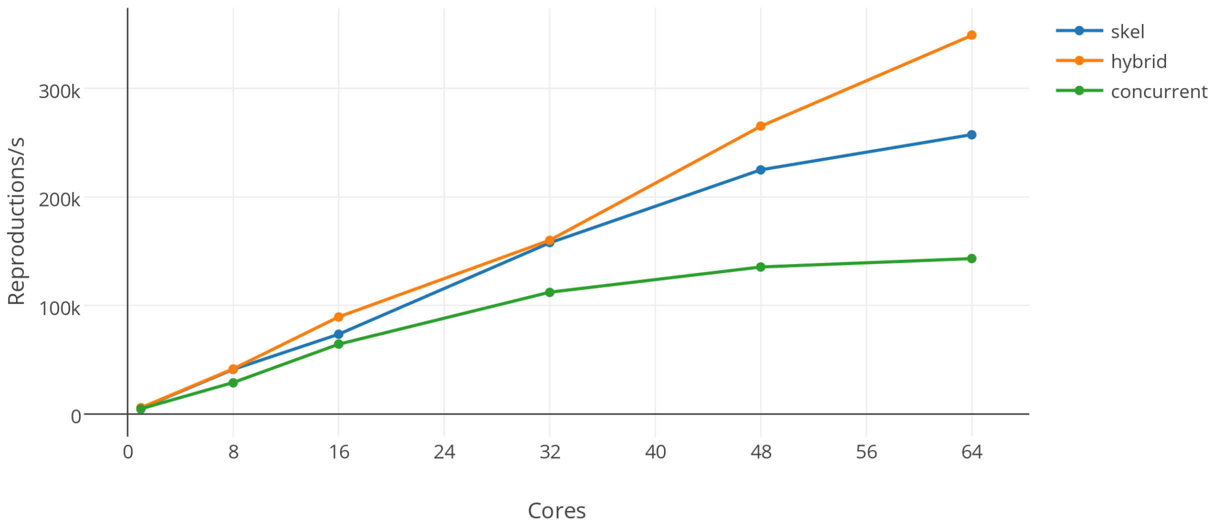

Figure 12. Performance after all optimizations

Unpredictable changes in the problem parameters can be addressed with approaches defined as *planning under uncertainty*. The methods assume that a planner does not have complete knowledge required to calculate a plan or the knowledge is uncertain. The solutions to this class of problems have to address an issue of uncertainty modeling [34]. An interesting example of a solution for mobile robot motion planning is presented in [35]. The planing algorithm handles sensing and motion uncertainty and optimizes motion plan with the complexity of $O[n^6]$ in a search space of n dimensions. Such methods cannot be applied for real-time motion planning when a group of independent vehicles is considered.

The problems of managing **entity groups** under uncertainty are also receiving attention, especially in the domain of mobile robotics. Tasks like formation control [36] or motion coordination [37] require robust planning and execution methods. In order to guarantee safety the solutions tend to simplify the planning algorithm or apply behavioural controllers, which can respond quickly in case of an unexpected execution error. This trade off between robustness and optimality of solutions cannot be overcome as long as the plan is prepared on-demand, when the unexpected situation has already occurred.

Planning in dynamic environments is also studied in the case of scheduling problems, e.g. Job Shop and similar ones, where it is necessary to receive new, unplanned jobs in certain time periods, and/or deal with potential machinery breakdowns. Such approach was usually called a rolling horizon procedure where a rolling time window is introduced and newly arriving jobs are included in the prediction window. Based on the predictions, schedules are prepared (subproblems of JSSP are solved) and final schedule is integrated in the current global solutions [38, 39]. In these cases, a shifting bottleneck heuristic is used for scheduling and rescheduling [40]. Such heuristics are of course very useful, and usually good-enough for the manual solving of such problems, but in the approach presented here, we would like rather to use a general-purpose evolutionary algorithms (in particular EMAS) to schedule the JSSP, however none of other possible heuristics are excluded and they may be considered in the future.

6.1. *Evolutionary Crossroads Lights Management*

Traffic management and in particular crossroads lights management problem can be defined as optimization one, and its high level of complexity and potential difficulty of solving (because of its highly dynamic nature) may be approached with general purpose metaheuristics, as evolutionary algorithms. In the case described here, traffic management of certain crossroads is encoded into the fitness function and optimized using EMAS. This is a first step towards further introduction of predictive multi-variant planning that will be one of our future work issues (preparing plans before they are needed, based on certain features of the earlier observed traffic, its participants, environmental conditions etc.).

In the described case, we assume discrete time and space and we use simplified cars motion model. Each car occupies a singe cell of a road lane and in each time step it moves to the succeeding cell it the cell is empty or becomes empty during the time step. A controlling element is located in the cells where two lanes cross. It can block one lane and enable the other. The plan for controlling the whole crossroad in a single time-step is a set of decisions concerning the controlling elements. The plan for controlling the whole crossroad for a period of time is a sequence of single-step plans.

Three steps of a plan executed in a two-lanes crossing are shown in Fig. 13. The plan is $(1, 1, 2)$, which means that during the first two time-steps only cars moving on lane1 can enter the crossing. In the third time-step (situation c) the plan enables motion of cars 3 and 4 moving on lane2, stopping car number 1.

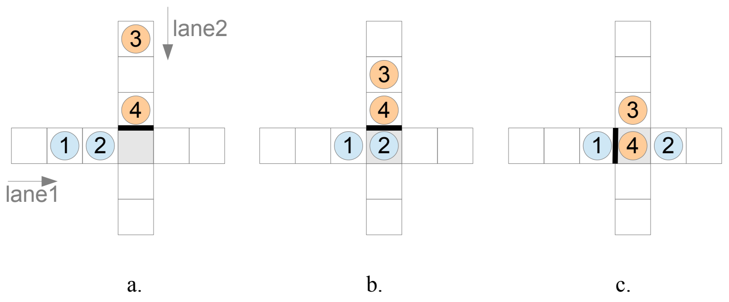

Figure 13. Three steps of a plan for two lanes crossing.

The efficiency of the plan is calculated as the number or cells covered by all the cars. Calculating the efficiency is a relatively complex task, which requires executing a simulation of cars according to a particular plan. Its complexity is $O(n^2 * t)$, where n is the number of cars and t is the number of time-steps in the plan.

The evolutionary algorithm used for optimizing the plan operates on individuals represented as $i = ((x_1^1, ..., x_m^1), ..., (x_1^t, ..., x_m^t))$, where m is the number of lanes crossings to control and $x_i^j \in \{1, 2\}$ determines which of the two lanes is blocked in the particular time-step. Each individual is therefore a sequence of $m * t$ bits, which makes variation operators definition straightforward. We use single-point crossover operation and bit-flip mutation. Each variation operator generates one or two new individuals (new plans) which must be evaluated. Multiple variants of possible future situations on the crossroad can be handled using more complex plan evaluation method. Assuming that each car can fail to reach desired location on time, we can generate several possible variants of the situation on the crossroad. Then the evaluation of a plan must involve calculating the efficiency of the plan in all v variants of situation, increasing the complexity to $O(n^2 * t * v)$. The total fitness should be calculated as a sum of fitness values for all variants. This approach promotes more universal plans, which do not cause significant loss of efficiency when slight differences in the situation occur.

6.2. *Scaling the Memory-intensive Operations*

The optimized version of the Erlang computing framework, presented in section 5, has been applied to the presented traffic planning problem. We used a single crossroad setup, which consisted of four lanes with four independent lane crossings, as presented in figure 14. The incoming lanes length was adjusted according to the number of cars, which varied from 10 to 100. Fitness value was calculated as the total number of steps covered by all the cars after 20 time-steps of the plan.

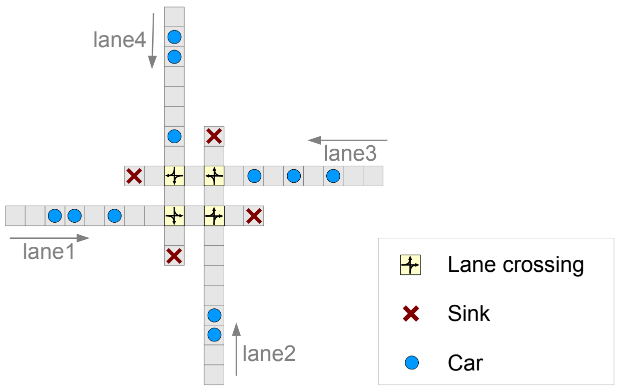

Figure 14. The crossroad layout used in the experiments.

The main difference from the previous use-case is the memory-intensive fitness function. Every fitness evaluation requires performing a entire traffic simulation, which involves intensive memory manipulation. This observation makes the choice of an appropriate data structure very important. Different Erlang built-in data structures can have a crucial influence on the program scalability and performance. Our initial approach used *dicts* as a major container for the simulation data, however we observed very poor performance and scalability. Fig. 15 compares the efficiency of different data structures.

Surprisingly, Erlang *proplists* are clearly the fastest solution for the use case. The reason might be that the structure itself is fairly small (one element per car, not more than 100), therefore the hashing functions used in other data structures are computationally more expensive than iterating over a small list. Another explanation could be that the smaller size and more predictable memory pattern of this data structure makes it more cache-friendly on multi-core hardware. It is also worth noting that *ETS* performed slightly better than *dicts*, but significantly worse than *gb_trees*. Only *proplists* and gb_trees allow linear scalability of the algorithm on a many-core architecture.

6.3. *Traffic Management Results*

The application of the algorithm in the problem of traffic management requires providing good solutions within specified time. Therefore the performance improvements are crucial for the considered application. Fitness convergence for different problem sizes computed on 64 cores is shown on figure 16.

The results show, that the method converges very fast, reaching stable solutions after 1-3 seconds, depending on problem size. The quality of found solutions is compared to the results of the fixed-cycles approach in table 1.

The system managed to calculate solutions better by 8%–40%, which is definitely a significant improvement of traffic efficiency. It seems that more complex crossroads with several lanes could benefit even more. The system

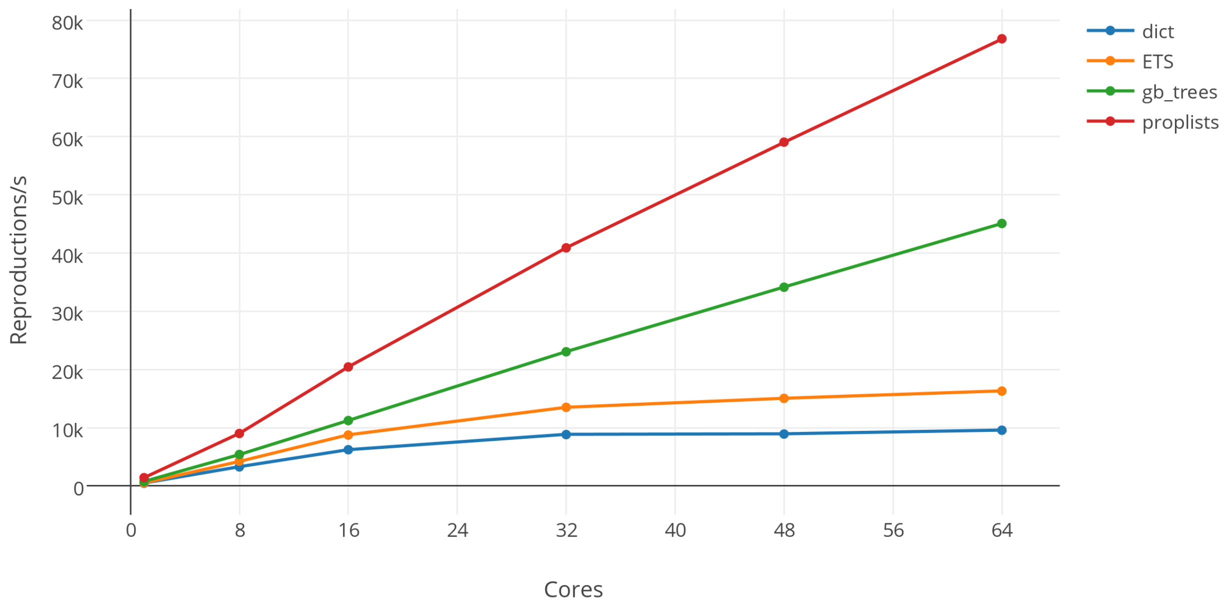

Figure 15. Performance of different data structures in traffic simulation using the concurrent model.

number of cars:	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
cycle each step	25	126	194	313	480	787	962	1047	1100	1347
cycle of 2 steps	25	112	194	396	522	753	962	1047	1164	1364
cycle of 5 steps	32	98	194	371	501	774	845	1025	1126	1347
plan after 3 seconds	42	138	260	456	612	852	1043	1263	1367	1627

Table 1. Comparison of traffic efficiency with fixed-cycles approach (traffic lights) and the evolutionary traffic management.

managed to provide very good solutions after time shorter than a second, which means that it should be able to handle larger problem sizes in reasonable time.

7. Conclusions

Metaheuristics, especially general-purpose ones are crucial methods for solving hard optimization problems. Agent-based metaheuristics turned out to be an effective tool for dealing with such problems, therefore looking for efficient way of implementation of agent-oriented frameworks becomes an important endeavour. Moreover, in the era of multi-core and many-core architectures, leveraging functional approach and scalable techniques may lead to reaching efficient, easy to use tools that can be adapted to many computing techniques and problems.

Erlang technology is frequently used for building large-scale systems successfully utilizing clusters of computers and multi-core processors in a reliable setting with very small maintenance overhead. However, the most ambitious (industrial) deployments of Erlang applications utilize 24 CPU cores, and even the creators of Erlang do not test the EVM on architectures consisting of more than 32 cores. Some academic research conducted shows results of scaling Erlang up to 64 cores and beyond (using e.g. Intel Xeon Phi architecture), but these reports are based mostly on low-level technology-oriented benchmarks.

Looking for new applications of the Erlang-based frameworks (like parallel computing), encourages to undertake new challenges to efficiently utilize existing supercomputing hardware—thus the approach to efficiently scale such a framework on 64 CPU core machine.

In this paper we presented the initial assumptions regarding the structure and the behaviour of the implemented system (population-based agent-oriented computing framework) pointing out the potential scalability problems and

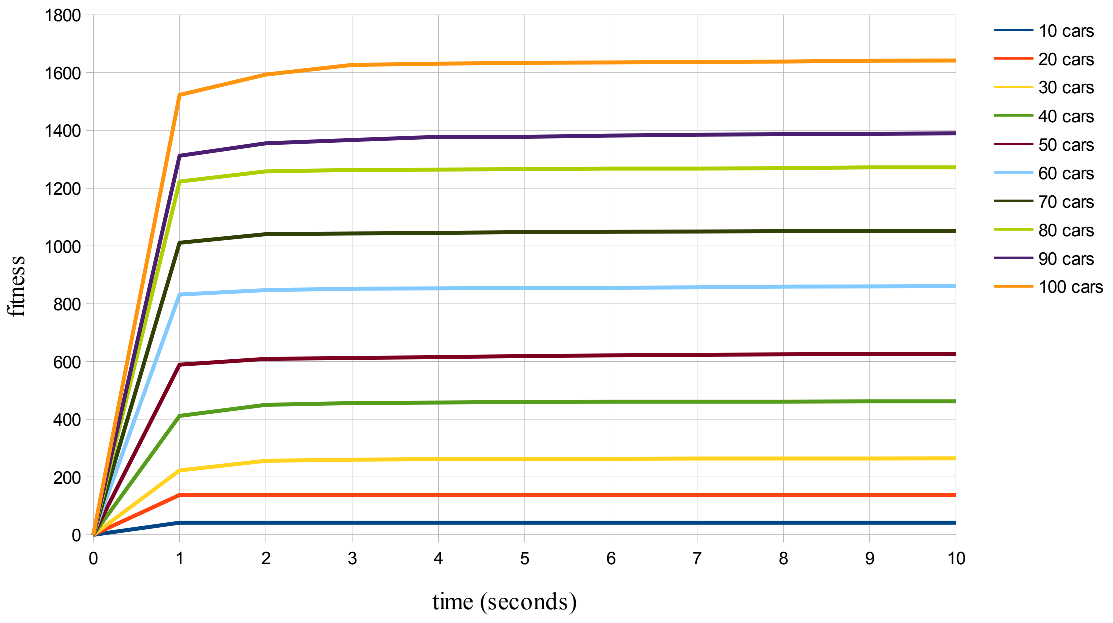

Figure 16. Fitness convergence for different problem sizes during 10 seconds

devising three versions of the framework. The procedure of profiling and refactoring led us to good scalability, beyond the acclaimed 24 CPUs. Starting from good scalability only on 8 CPU cores, the search for bottlenecks, monitoring of schedulers' load with predefined tools and search for locks in the running code were performed. As a result of this research, the existing framework was appropriately modified, and in the end, the assumed scalability was reached, starting from the benchmark optimization (Rastrigin function) and finishing on a far more complex optimization problem (traffic management).

The main conclusion of the presented work is that it is possible to implement a computationally intensive applications in Erlang, which scale up to 64 cores, however, achieving this is not a straightforward process. We are convinced, that our experience gathered in the course of this research may be utilised by other researchers and developers preparing highly scalable computation-intensive frameworks or applications based on the Erlang technology. In the future we plan to extend our knowledge by coping with computing in a heterogeneous hardware environment using the available computing frameworks.

In the future, from the technological point of view, we are planning to try to further scale our platform using Intel Xeon Phi architecture, and from the substantial point of view, we are going to further explore the multi-variant planning problem, treating the results presented in this paper as a good basis and encouragement for its efficient implementations.

Acknowledgements

The research presented in the paper was conducted using PL-Grid Infrastructure (http://www.plgrid.pl/en). The research presented in the paper has received founding from the Polish National Science Centre under grant DEC-2011/01/D/ST6/06146. The authors would like to express heartfelt gratitude to a member of the OTP team, Lukas Larsson, for his advices on profiling Erlang code.

References

[1] K. Cetnarowicz, M. Kisiel-Dorohinicki, E. Nawarecki, The application of evolution process in multi-agent world (MAW) to the prediction system, in: M. Tokoro (Ed.), Proc. of the 2nd Int. Conf. on Multi-Agent Systems (ICMAS'96), AAAI Press, 1996.

[2] A. Byrski, M. Kisiel-Dorohinicki, E. Nawarecki, Agent-based evolution of neural network architecture, in: M. Hamza (Ed.), Proc. of the IASTED Int. Symp.: Applied Informatics, IASTED/ACTA Press, 2002.

[3] G. Dobrowolsi, M. Kisiel-Dorohinicki, E. Nawarecki, Some approach to design and realisation of mass multi-agent systems, in: R. Schaefer, S. Sedziwy (Eds.), Advances in Multi-Agent Systems, Jagiellonian University, 2001.

[4] A. Byrski, R. Schaefer, M. Smołka, Asymptotic guarantee of success for multi-agent memetic systems, Bulletin of the Polish Academy of Sciences—Technical Sciences 61 (1).

[5] L. Siwik, R. Dreżewski, Agent-based multi-objective evolutionary algorithms with cultural and immunological mechanisms, in: W. P. dos Santos (Ed.), Evolutionary computation, In-Teh, 2009, pp. 541–556.

[6] A. Byrski, M. Kisiel-Dorohinicki, Immunological selection mechanism in agent-based evolutionary computation, in: M. A. Klopotek, S. T. Wierzchon, K. Trojanowski (Eds.), Intelligent Information Processing and Web Mining : proceedings of the international IIS: IIPWM '05 conference : Gdansk, Poland, Advances in Soft Computing, Springer Verlag, 2005, pp. 411–415.

[7] K. Wróbel, P. Torba, M. Paszyński, A. Byrski, Evolutionary multi-agent computing in inverse problems, Computer Science (accepted for printing).

[8] A. Byrski, Tuning of agent-based computing, Computer Science (accepted).

[9] J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High-Performance Programming, Elsevier Science, 2013.

[10] A. Varghese, B. Edwards, G. Mitra, A. Rendell, Programming the adapteva epiphany 64-core network-on-chip coprocessor, in: Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, 2014, pp. 984–992.

[11] EZchipSemiconductor, Highest core-count arm processor optimized for high performance networking applications, `http://www.tilera.com/products/?ezchip=585\&spage=686` (04 2015).

[12] F. Bellifemine, A. Poggi, G. Rimassa, Jade: A fipa2000 compliant agent development environment, in: Proceedings of the Fifth International Conference on Autonomous Agents, AGENTS '01, ACM, New York, NY, USA, 2001, pp. 216–217. doi:10.1145/375735.376120.
URL `http://doi.acm.org/10.1145/375735.376120`

[13] A. Pokahr, L. Braubach, W. Lamersdorf, Jadex: Implementing a bdi-infrastructure for jade agents, EXP—In Search of Innovation (Special Issue on JADE) 3 (1) (2003) 76–85.

[14] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, G. Balan, Mason: A multiagent simulation environment, Simulation 81 (7) (2005) 517–527. doi:10.1177/0037549705058073.
URL `http://dx.doi.org/10.1177/0037549705058073`

[15] M. North, N. Collier, J. Ozik, E. Tatara, C. Macal, M. Bragen, P. Sydelko, Complex adaptive systems modeling with repast simphony, Complex Adaptive Systems Modeling 1 (1) (2013) 3. doi:10.1186/2194-3206-1-3.
URL `http://www.casmodeling.com/content/1/1/3`

[16] D. Thomas, Functional programming–crossing the chasm, Journal of object technology.

[17] F. Cesarini, S. Thompson, Erlang programming, O'Reilly Media, Inc., 2009.

[18] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, I. E. Venetis, A scalability benchmark suite for erlang/otp, in: Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop, Erlang '12, ACM, New York, NY, USA, 2012, pp. 33–42. doi:10.1145/2364489.2364495.

[19] K. Sagonas, K. Winblad, More scalable ordered set for ets using adaptation, in: Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang, Erlang '14, ACM, New York, NY, USA, 2014, pp. 3–11. doi:10.1145/2633448.2633455.

[20] S. Zheng, X. Long, J. Yang, Using many-core coprocessor to boost up erlang vm, in: Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang, Erlang '13, ACM, New York, NY, USA, 2013, pp. 3–14. doi:10.1145/2505305.2505307.

[21] D. Krzywicki, L. Faber, A. Byrski, M. Kisiel-Dorohinicki, Computing agents for decision support systems, Future Generation Comp. Syst. 37 (2014) 390–400. doi:10.1016/j.future.2014.02.002.
URL `http://dx.doi.org/10.1016/j.future.2014.02.002`

[22] E. Cantú-Paz, A survey of parallel genetic algorithms, Calculateurs Paralleles, Reseaux et Systems Repartis 10 (2) (1998) 141–171.

[23] S. Russel, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 2003.

[24] R. Sarker, T. Ray, Agent-Based Evolutionary Search, 1st Edition, Vol. 5 of Adaptation, Learning and Optimization, Springer, 2010.

[25] S.-H. Chen, Y. Kambayashi, H. Sato, Multi-Agent Applications with Evolutionary Computation and Biologically Inspired Technologies, IGI Global, 2011.

[26] P. Uhruski, M. Grochowski, R. Schaefer, Multi-agent computing system in a heterogeneous network, in: Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC 2002), IEEE Computer Society Press, Warsaw, Poland, 2002, pp. 233–238.

[27] J. Liu, Y. Tang, Y. Cao, An evolutionary autonomous agents approach to image feature extraction, Evolutionary Computation, IEEE Transactions on 1 (2) (1997) 141–158. doi:10.1109/4235.687881.

[28] J. Liu, H. Jing, Y. Tang, Multi-agent oriented constraint satisfaction, Artificial Intelligence 136 (1) (2002) 101 – 144. doi:http://dx.doi.org/10.1016/S0004-3702(01)00174-6.
URL `http://www.sciencedirect.com/science/article/pii/S0004370201001746`

[29] E. Alba, B. Dorronsoro, Cellular Genetic Algorithms, Springer, 2008.

[30] A. Byrski, R. Drezewski, L. Siwik, M. Kisiel-Dorohinicki, Evolutionary multi-agent systems, The Knowledge Engineering Review 30 (2015) 171–186.

[31] D. Krzywicki, L. Faber, A. Byrski, M. Kisiel-Dorohinicki, Computing agents for decision support systems, Future Generation Comp. Syst. 37 (2014) 390–400.

[32] V. Janjic, A. Barwell, K. Hammond, Using erlang skeletons to parallelise realistic medium-scale parallel programs, Proceedings of the Workshop on High-Level Programming for Heterogeneous and Hierarchical Parallel Systems.

[33] J. Digalakis, K. Margaritis, An experimental study of benchmarking functions for evolutionary algorithms, International Journal of Computer Mathemathics 79 (4) (2002) 403–416.

[34] J. Mula, R. Poler, J. García-Sabater, F. Lario, Models for production planning under uncertainty: A review, International Journal of Production

Economics 103 (1) (2006) 271 – 285.
[35] J. Van Den Berg, S. Patil, R. Alterovitz, Motion planning under uncertainty using iterative local optimization in belief space, The International Journal of Robotics Research 31 (11) (2012) 1263–1278.
[36] T. Balch, C. Arkin, Behavior-based Formation Control for Multi-robot Teams, in: IEEE Transactions on Robotics and Automation, 1997, pp. 926–939.
[37] W. Turek, K. Cetnarowicz, W. Zaborowski, Software agent systems for improving performance of multi-robot groups, Fundam. Inf. 112 (1) (2011) 103–117.
[38] J. Fang, Y. Xi, A rolling horizon job shop rescheduling strategy in the dynamic environment, The International Journal of Advanced Manufacturing Technology 13 (3) (1997) 227–232.
[39] B. Wang, Q. Li, Rolling horizon procedure for large-scale job-shop scheduling problems, in: Automation and Logistics, 2007 IEEE International Conference on, 2007, pp. 829–834. doi:10.1109/ICAL.2007.4338679.
[40] M. Pinedo, Planning and Scheduling in Manufacturing and Services, Springer, 2009.

***Highlights (for review)**

- agent-oriented platform for metaheuristic computing in Erlang
- scaling of the platform up to 64 cores (exceeding many similar approaches)
- leveraging of current hardware architectures by the means of functional programming for computing
- simple use case: classic benchmark optimization problem - multi-modal Rastrigin function
- complex use case: evolutionary urban traffic planning, memory-intensive fitness function
- agent-based metaheuristics used: Evolutionary Multi-Agent System (EMAS)