Procedia Computer Science

Volume 29, 2014, Pages 1068–1077

ICCS 2014. 14th International Conference on Computational Science

Generation-free Agent-based Evolutionary Computing

D. Krzywicki, J. Stypka, P. Anielski, Ł. Faber,
W. Turek, A. Byrski, and M. Kisiel-Dorohinicki

AGH University of Science and Technology
{daniel.krzywicki,jasieek,anielski,faber,turek,olekb,doroh}@agh.edu.pl

Abstract
Metaheuristics resulting from the hybridization of multi-agent systems with evolutionary computing are efficient in many optimization problems. Evolutionary multi-agent systems (EMAS) are more similar to biological evolution than classical evolutionary algorithms. However, technological limitations prevented the use of fully asynchronous agents in previous EMAS implementations. In this paper we present a new algorithm for agent-based evolutionary computations. The individuals are represented as fully autonomous and asynchronous agents. Evolutionary operations are performed continuously and no artificial generations need to be distinguished. Our results show that such asynchronous evolutionary operators and the resulting absence of explicit generations lead to significantly better results. An efficient implementation of this algorithm was possible through the use of Erlang technology, which natively supports lightweight processes and asynchronous communication.

Keywords:

1 Introduction

Different evolutionary algorithms use specific representations, variation operators, and selection schemes. However, they all employ a similar model of evolution: they work on a number of data structures (*populations*) and repeat in cycles (*generations*) the same process of selecting parents and producing offspring using variation operators. This simplified model of evolution lacks several properties observed in organic evolution, such as: a) dynamically changing environmental conditions, b) a dependency on multiple criteria, c) the co-evolution of species, d) the evolution of the genotype-fenotype mapping, e) the assumption of neither global knowledge nor generational synchronisation.

Evolutionary Multi-Agent Systems (EMAS). Over the last decade, our group has worked to overcome these shortcomings by developing the idea of decentralised evolutionary computations [2] in the form of evolutionary multi-agent systems [5]. EMAS are a hybrid meta-heuristic

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.096

which combines multi-agent systems with evolutionary algorithms. The idea consists in evolving a population of agents to improve their ability to solve a particular problem.

In a multi-agent system no global knowledge is available to individual agents [9]. Agents should remain autonomous and no central authority should be needed. Therefore, in 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.

Selection in an EMAS is designed so that agents with good behaviour become more likely to reproduce. What exactly good behaviour means and how it affects the likelihood of reproduction depends on the problem being solved and on implementation details. So far, many variants have been proposed, combining the general idea described above with concepts such as immunological, elitist or co-evolutionary EMAS (for details refer to [2]).

These systems were applied to different problems (global, multi-criteria and multi-modal optimization in continuous and discrete spaces), and the results clearly showed superior performance in comparison to classical approaches (see e.g., [3, 12, 4, 2]).

Agent-oriented systems are asynchronous by nature. During the last years, we made several approaches to construct efficient software targeted at variants of EMAS and at agent-based computing in general. There is a number of popular agent-oriented frameworks which offer asynchronously communicating agents (such as Jadex [13], Jade [1] or MadKit[7]). However, they all share similar properties, such as heavyweight agents, at least partial FIPA-compliancy (JADE) or a BDI model (Jadex). It is also common for each agent to be executed as a separate thread (e.g. in JADE). These traits are indeed appropriate to model flexible, coarse-grained, open systems. However, evidence suggest they are not best suited for closed systems with relatively homogeneous agents nor for fine-grained concurrency with large numbers of lightweight agents, which are both common in evolutionary computations [14].

Therefore, we had to construct dedicated tools. We first focused on the simulation of decentralized agent behaviour and prepared several fully synchronous versions which resulted in the implementation of the AgE platform[1]. We applied a phase-model of simulation in this platform, efficiently implementing such EMAS aspects as decentralized selection, supporting the user with different component-oriented utilities, increasing reuse possibilities and allowing flexible configuration of the computing system.

Generation-free EMAS. In this paper, we present a promising new approach to these kinds of algorithms. We used Erlang lightweight processes to implement a fined grained multi-agent system. Agents are fully asynchronous and autonomous in fulfilling their goals, such as exchanging resources with others, reproducing or being removed from the system. Agents are able to coordinate their behaviour with the use of mediating entities called meeting arenas.

This approach brings us closer to the biological origins of evolutionary algorithms by removing artificial generations imposed by step-based implementations. We show the concept of generation-oriented and generation-free computing in Fig. 1.

The first case (a) shows the classical approach, which consists of transforming a population of individuals with a stochastic function. This function is usually the composition of some predefined operators, such as selection, crossover or mutation. The second case (b) illustrates the EMAS approach (as in AgE), where different transformation functions may be applied as the results of agent actions. This model still assumes the existence of generations, as a result of a step-based simulation. In fact, both above cases use discrete-time based simulation.

[1] http://age.agh.edu.pl

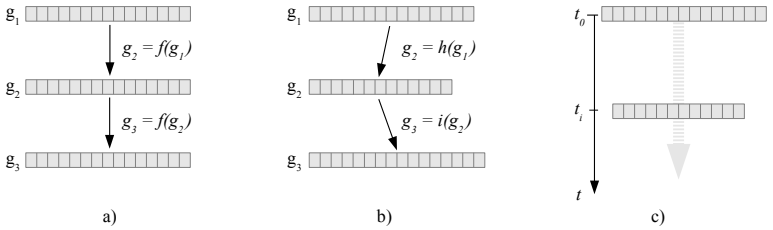

Figure 1: Concept of generation-oriented and generation-free evolutionary computing.

In contrast, the third case (c) is a nearly continuous-time simulation (if we disregard the nature of the computing machine). In this model, all the agents may initiate actions in any possible time. The Erlang scheduler makes sure they are given computing resources when they need them.

Functional Agent Systems. Until now, there was little number of multi-agent systems created according to the functional paradigm. The most mature one is eXAT (*erlang eXperimental Agent Tool*) [6]. It was presented in 2003 and implemented in Erlang. The goal of this platform, as stated by its authors, was to test the feasibility of using functional programming languages as a development tool for FIPA-compliant agents.

An agent in eXAT is an actor-based and independent entity composed of *ERES engines* and *behaviours*. The former are rule processing systems based on the ERES library[2]. These engines have *rules* and *knowledge bases* and represent the 'mental state' of the agent.

Behaviours describe the functionality of the agent as a finite-state machine. Transitions are triggered by changes to the facts or by external events in the form of messages. Each transition represents an action of the agent. The original version of eXAT does not support agent migrations, however there are forks that introduced such functionality [11].

The rest of the paper is structured as follows: In section 2 we describe the design of our multi-agent system. The main contribution of this paper lies in the fully asynchronous nature of the simulation, which is achieved through the use of Erlang technology and the concept of meeting arenas. The efficiency of the algorithm is presented and discussed along with the experimental results in section 3. We conclude the paper by showing possible opportunities for future work.

2 System Design

The multi-agent system described in this work has been implemented in Erlang, as the lightweight concurrency provided by Erlang processes is well suited to model both individual agents as whole populations. As our multi-agent system has a specific purpose and do not need to communicate with the outside world, compliance with FIPA standards is not required. Instead, we focus on the computational model, available parallelism and efficiency.

The following part of this section describes the algorithm used in our evolutionary multi-agent system, the model of agents interactions and three alternative implementations. In contrast with eXAT, agents in our system are simpler and tailored for an evolutionary algorithm. However, our design could also be extended to more complex use cases.

[2]Seresye is used in newer versions, available at: `https://github.com/tenio/seresye`.

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 [2]. 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, 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.

As in other evolutionary algorithms, agents can be split into separate populations. Such *islands* help preserve diversity by introducing allopatric speciation and can also execute in parallel. Information is exchanged between islands through agent migrations.

Meeting arenas. The remaining important issue is how to efficiently design and implement meetings between agents. The decisions at that stage have an impact on the properties of the algorithm, on its computational efficiency and on its potential for parallelism.

Agents can be put on a lattice so that only agents on the same node meet. However, depending on the number of nodes and the number of agents, we can be back to the problem of multiple meetings on a single node.

A general and simple way to perform meetings is to randomly shuffle the list of agents and then process each pair sequentially or in parallel. This approach has several limitations:

- The whole population needs to be gathered in order to shuffle agents and group them in pairs. This is unfortunate in an algorithm which should be decentralized by nature.
- Agents wanting to perform different actions (e.g. fight or reproduction) may be grouped together nevertheless. Therefore, all combinations of possible desired behaviours need to be handled in every meeting.

In a previous work [10], we proposed to group agents willing to perform similar actions in separate entities called *meeting arenas*, following the Mediator design pattern. Agents choose and join an arena depending on their state (e.g. their amount of energy). Arenas split incoming agents into groups of some given cardinality and trigger the actual meetings (see Fig. 2). Every kind of agent behaviour is represented by a separate arena.

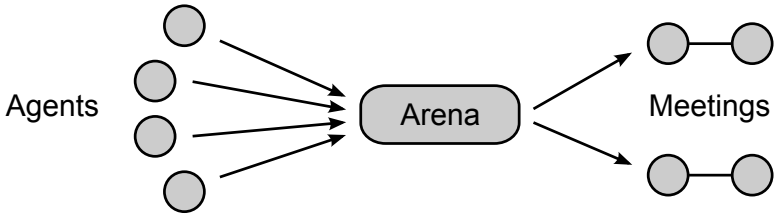

Figure 2: Meeting arenas allow to group similar agents and coordinate meetings between them.

The dynamics of the multi-agent system are fully defined by two functions. The first function represents agent behaviour and chooses an arena for each agent (mapping step). The second function represents meeting logic and is applied in every arena (reducing step). This approach is similar to the MapReduce model and has the advantage of being very flexible, as it can be implemented in both a centralized and synchronous way or a decentralized and asynchronous one, as we show further below.

Sequential implementation. The sequential version of our multi-agent system is implemented as a discrete event simulation. In every step, the *behaviour* function (see Listing 1) divides the population into partitions corresponding to available arenas.

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

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

Partitions are further subdivided into pairs and processed by applying a different meeting function, depending on the type of arena (see. Listing 2). Every meeting yields a resulting group of agents, some of which might have had their state changed (e.g. by transferring energy) or might have been created (e.g. as a result of reproduction). Finally, some agents may be removed from the output, as in the death arena. All resulting groups are then merged in order to form the new population, which is shuffled before the next step.

```
1 meeting(death, Agents) -> []
2 meeting(reproduction, Agents) ->
3   lists:flatmap(
4     fun doReproduce/1,
5     inGroupsOf(2, Agents));
6 meeting(fight, Agents) ->
7   lists:flatmap(
8     fun doFight/1,
9     inGroupsOf(2, Agents));
```

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

Multiple islands can be represented as separate lists of agents, each of which is processed in turn. Migration is performed at the end of each step by moving some agents between lists.

Hybrid implementation. It is easy to introduce coarse-grained concurrency in the multi-agent system. In our second implementation, every island is managed by a separate Erlang process, responsible for executing in a loop the sequential algorithm described above. Islands communicate through message-passing, no other synchronization is performed.

Agent migration is realized in a different way than in the sequential implementation. The behaviour function from Listing 1 is modified by adding a migration action which is chosen with some low probability. A migration arena is also introduced on every island.

The migration arena removes any incoming agent from the population and sends it as a message to a special migration process. Upon receipt, this process forwards the agent to an island chosen according to some topology. In every step, the processes responsible for executing islands empty their mailbox and incorporate the incoming agents into their population.

Concurrent implementation. Finer-grained concurrency can be achieved by modelling every agent and every arena as a separate Erlang process which communicates with the outside world only through message passing. The algorithm becomes fully asynchronous, meaning that every agent acts at its own pace and there is no population-wide step. Meeting arenas become especially useful in this implementation, as they greatly simplify communication protocols.

Every agent joins an arena, chosen as before, by sending it a message containing some of its state. As soon as enough agents gather in it, the arena triggers a meeting. As a result, new agents may be created, existing agents may be killed or replied to with a message updating their state. Upon receiving such an acknowledgement, agents choose an arena again and the process is repeated.

Islands can be logically defined simply as distinct sets of arenas (Fig. 3), as agents can only meet with other agents which share the same set of arenas. Therefore, migrating an agent simply means changing the arenas it meets on. Although we do not examine it in this work, the model could be extended to have some areas shared by several islands or multiple arenas of the same kind within a single island.

Fights and reproductions arenas behave just as before. Meeting arenas choose an island according to some topology, lookup the corresponding arenas and send their addresses back to the agent. The agent updates the set of arenas available to itself and resumes its behaviour. As it will now be able to meet with a different set of agents, it has indeed migrated.

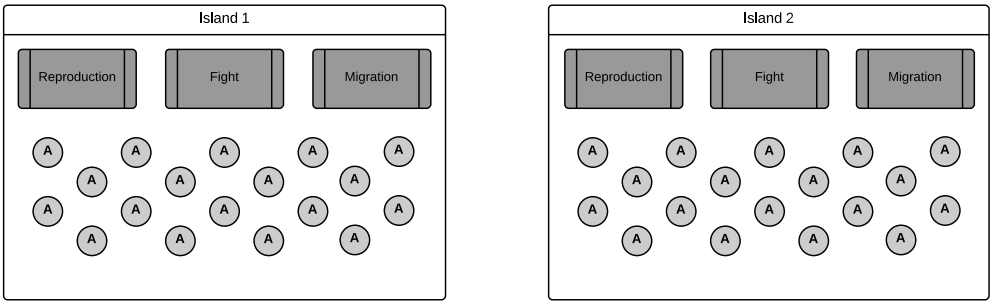

Figure 3: The structure of the concurrent model - agents meet on arenas within islands, which are simply distinct sets of arenas. Migrating an agent does not change its location - instead, it changes the set of arenas the agent communicates with.

3 Methodology and Results

We used our multi-agent system to minimize the Rastrigin function, a common benchmarking function used to compare evolutionary algorithms. This function is highly multimodal with many local minima and one global minimum equal 0 at $\bar{x} = 0$. We used a problem size (the dimension of the function) equal to 100, in a domain equal to the hypercube $[-50, 50]^{100}$. One particular property of the Rastrigin function is that values lower than 1 correspond to the basin of attraction of the global optimum.

We ran our simulations on Intel Xeon X5650 nodes provided by the Pl-Grid[3] infrastructure at the ACC Cyfronet AGH[4]. We used up to 12 cores and 1 GB of memory.

We tested the 3 implementations described in the previous section. The hybrid and concurrent models were run on 1,2,4,8 and 12 cores. The sequential version was only run on 2 cores, as initial experiments did not show significant improvements beyond that number (the second core allowed faster garbage collecting, logging, etc.). Every experiment was repeated 30 times in order to obtain statistically significant results. The results further below are averaged over these 30 runs.

[3]http://www.plgrid.pl/en
[4]http://www.cyfronet.krakow.pl/en/

The hybrid model does not benefit from a number of cores higher than the number of islands. As we had 12 cores at our disposal, we used 12 islands in every experiment. Migration destinations were chosen at random in a fully connected topology.

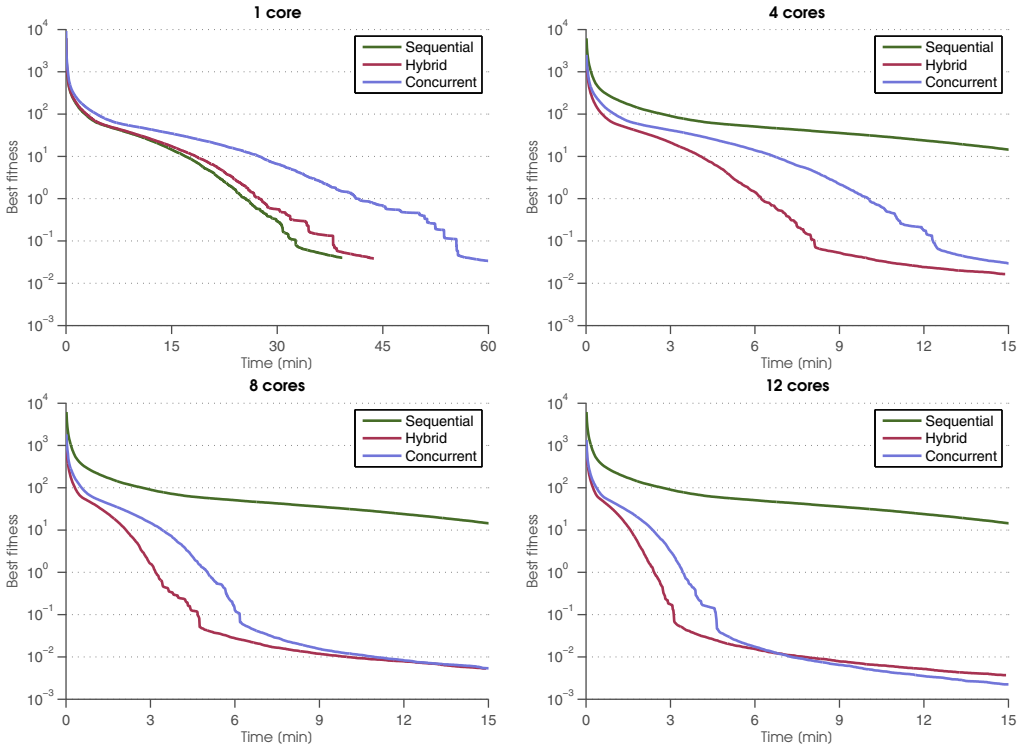

Figure 4: Best fitness ever over time, depending on the model and the number of cores.

Results. We examined our models under two criteria: how well the algorithm works and how fast the meeting mechanism is.

We assessed the quality of the algorithm by recording: the fitness of the best solution found so far at any given time (see Fig. 4) and the *first time to hit*, that is the time needed to reach the attraction basin of the global solution, i.e. fitness values lower than 1 (see Fig. 5 left).

We estimated the speed of the models by counting the amount of agent meetings performed in a unit of time. These numbers appeared to be proportionally related across different arenas. Therefore, we only considered the amount of reproductions per second (see Fig. 5 right), which also gives an indication on the number of fitness function evaluations.

Finally, we checked the scalability of the models by computing the absolute and relative speedup of both the first time to hit and the number of total reproductions (see Fig. 6).

Discussion Of course, the sequential algorithm does not improve when more cores are used. The hybrid one can, however, be perceived as its simple generalization. In fact, both perform similarly on a single core, the lack of communication overhead making the sequential version slightly better. The hybrid version improves when cores are added (Fig. 4).

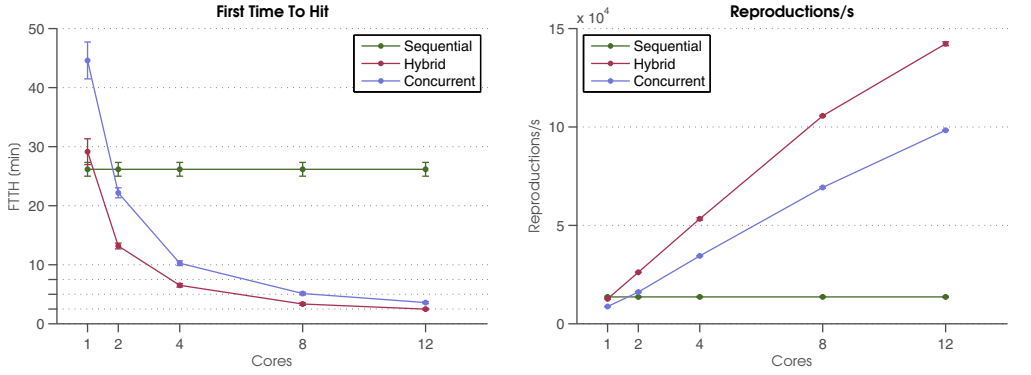

Figure 5: First time to hit and number of reproductions per second, depending on the model and the number of cores (with 95% confidence intervals).

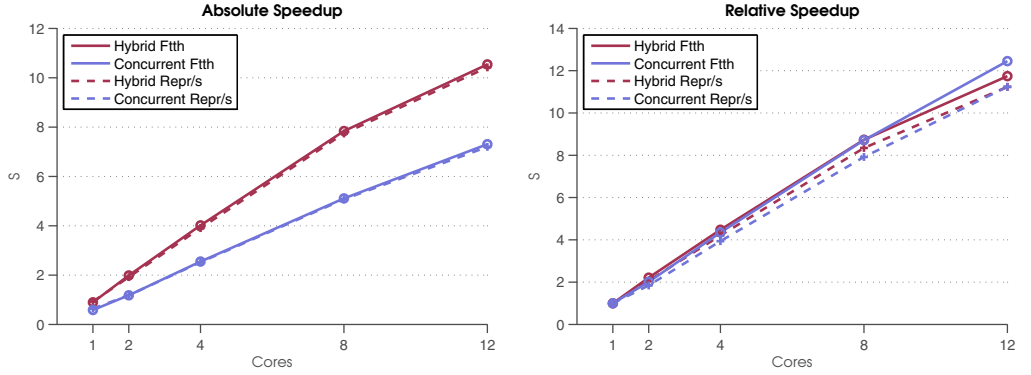

Figure 6: Absolute and relative speedup of each of the models, in terms of first time to hit and number of reproductions per second.

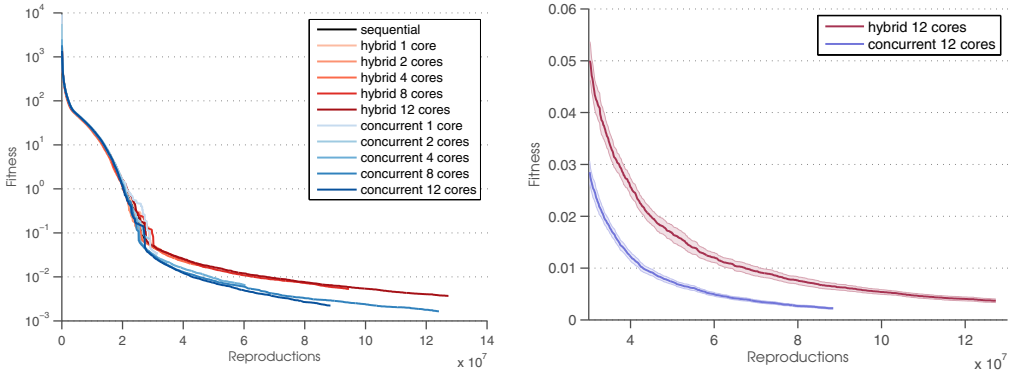

Figure 7: Best fitness ever over *the total number of reproductions*. The right image shows a close-up on the average tails, along with 95% confidence intervals, for 12 cores.

As the number of cores increase, the concurrent version catches up to the hybrid one (Fig. 4). Above 8 cores, there is no statistically significant difference in the first time to hit (Fig.

5 left). Preliminary experiments showed that the concurrent version still improves when the number of cores is bigger than the number of islands, in contrast to the hybrid one.

The major difference between the results for both models is the number of reproduction happening every second (Fig. 5 right). These numbers increase nearly linearly with the increase of nodes. However, they are smaller in the case of the concurrent version.

In fact, the concurrent version scales nearly linearly, as shown by its speedups in Fig. 6, while the hybrid version start to saturate at a higher number of cores..

The interpretation of the smaller number of agent reproductions in the concurrent version can be twofold: on one hand, they may indicate that the concurrent implementation is slow at processing agent meetings. On the other hand, as the number of reproductions reflect the number of fitness evaluations, less of those are needed in order to reach a similar solution.

The latter becomes all the more evident when plotting the best fitness over the number of total reproductions instead of over time. Fig. 7 left confirms that the characteristics of each version is not dependent on the number of cores used. The concurrent version distinguishes itself by a much lower number of reproductions and therefore less function evaluations, especially in the later phase of the computation (Fig. 7 right). We confirmed the statistical significance of this difference using a two-sample Kolmogorov–Smirnov test with $p = 0.05$.

This difference in dynamics could be explained in the following way: in the hybrid version, agents in the population are effectively synchronized, in the sense that all fights and all reproductions in a step need to end for any agent to move on. In contrast, in the concurrent version fights happen independently of reproduction and the population evolves in a much more continuous way. Information spreads faster in the population and the solution can be found with fewer generations.

Therefore, as the concurrent version needs less function evaluation, we conjecture that it should perform even better compared to the hybrid one when faced with real-life problems, where the computation of the fitness function itself can take much time.

4 Conclusions

The generation-free evolutionary multi-agent system presented in this paper gives very promising results in terms of scalability and efficiency. Its asynchronous nature allows to express behaviours more similar to the mechanisms observed in biological evolution, going beyond the classical generation-based approach. We applied this algorithm to a typical optimization problem. The results of our experiments indicate that when many agents are involved, this concurrent model is equally fast but more efficient in terms of function evaluations than the classical island model. An efficient implementation of this algorithm was possible because of the features of the Erlang technology, like lightweight processes and fast message passing concurrency. A further development of this method on modern multi-core supercomputers using Erlang [8] seems a promising direction of research. In particular, we are currently working on improving the concurrency of this algorithm even further and testing its scalability on more cores. Future research could also determine how this algorithm behaves on optimization problems with computationally-intense fitness functions. Finally, as the competitiveness of this method is visible when many agents and thus many islands are used, an interesting question is how the number and connectivity of the islands will affect the efficiency of the algorithm.

Acknowledgement

The research presented in the paper was partially supported by the European Commission FP7 through the project ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, under contract no.: 288570 (http://paraphrase-ict.eu). The research presented in this paper received partial financial support from AGH University of Science and Technology statutory project and AGH Faculty of Computer Science, Electronics and Telecommunications Dean's grant for young scientists. The research presented in the paper was conducted using PL-Grid Infrastructure (http://www.plgrid.pl/en).

References

[1] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade: A fipa2000 compliant agent development environment. In *Proceedings of the Fifth International Conference on Autonomous Agents*, AGENTS '01, pages 216–217, New York, NY, USA, 2001. ACM.

[2] A. Byrski, R. Dreżewski, L. Siwik, and M. Kisiel-Dorohinicki. Evolutionary multi-agent systems. *The Knowledge Engineering Review*, 2013 (accepted for printing).

[3] A. Byrski, W. Korczyński, and M. Kisiel-Dorohinicki. Memetic multi-agent computing in difficult continuous optimisation. In *Proceedings of 6th International KES Conference on Agents and Multi-agent Systems Technologies and Applications, 2013, Hue City, Vietnam, IOS Press (accepted in 2013)*. Springer.

[4] Aleksander Byrski. Tuning of agent-based computing. *Computer Science (accepted)*, 2013.

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

[6] Antonella Di Stefano and Corrado Santoro. Supporting agent development in Erlang through the eXAT platform. In *Software Agent-Based Applications, Platforms and Development Kits*, pages 47–71. Springer, 2005.

[7] Olivier Gutknecht and Jacques Ferber. The madkit agent platform architecture. *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, 2001.

[8] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. Gonzalez-Velez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer. The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems. In *FMCO: 10th International Symposium on Formal Methods for Components and Objects-Revised Selected Papers*, volume 7542, pages 218–236. Springer LNCS, 2013.

[9] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[10] D. Krzywicki, Ł. Faber, A. Byrski, and M. Kisiel-Dorohinicki. Computing agents for decision support systems. *Future Generation Computer Systems*, 2014.

[11] M. Piotrowski and W. Turek. Software Agents Mobility Using Process Migration Mechanism in Distributed Erlang. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 43–50, New York, NY, USA, 2013. ACM.

[12] S. Pisarski, A. Rugała, A. Byrski, and M. Kisiel-Dorohinicki. Evolutionary multi-agent system in hard benchmark continuous optimisation. In *Proc. of EVOSTAR Conference, Vienna*. IEEE (accepted for printing), 2013.

[13] Alexander Pokahr, Lars Braubach, and Kai Jander. The Jadex Project: Programming Model. In *Multiagent Systems and Applications*. 2013.

[14] Wojciech Turek. Erlang as a high performance software agent platform. *Advanced Methods and Technologies for Agent and Multi-Agent Systems*, 252:21, 2013.