Exploring concurrent and stateless evolutionary algorithms*

Subtitle†

Anonymous Author 1
Anonymous institute 1
Dublin, Ohio
aa1@ai1.com

Anonymous Author 2
Anonymous institute 2
Dublin, Ohio
2aa@ai2.com

Anonymous Author 3
Anonymous institute 2
Dublin, Ohio
2aa@ai2.com

Anonymous Author 4
Anonymous institute 2
Dublin, Ohio
2aa@ai2.com

ABSTRACT

Concurrent algorithms use channels for communication, which implies that communication is an integral part of them, so some attention must be devoted to its design. In the design of concurrent evolutionary algorithms, there are several options that can be used for performing this communication. In this paper we will explore how communication overhead can be reduced, and how it influences scaling. The evolutionary algorithm will use a concurrent language, and leverage its capabilities. Eventually, we will try to prove how concurrent version of algorithms offer a good option to leverage the multi-threaded and multi-core capabilities of modern computers.

CCS CONCEPTS

• **Theory of computation → Concurrent algorithms**; • **Computing methodologies → Genetic algorithms**; • **General and reference →** *Performance*;

KEYWORDS

Concurrency, Concurrent evolutionary algorithms, performance evaluation, algorithm design

ACM Reference Format:

Anonymous Author 1, Anonymous Author 2, Anonymous Author 3, and Anonymous Author 4. 2019. Exploring concurrent and stateless evolutionary algorithms: Subtitle. In *Proceedings of the Genetic and Evolutionary Computation Conference 2019 (GECCO '19).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Despite the emphasis on hardware-based techniques such as cloud computing or GPGPU, there are not many papers [15] dealing with creating concurrent evolutionary algorithms that work in a single computing node or that extend seamlessly from single to many computers.

The concurrent programming paradigm (or concurrency oriented programming [2]) is characterized by the presence of programming constructs for managing processes like first-class objects. That is, with native operators for acting upon them and the possibility of using them as parameters or as return values of a function. The latter yields to changes in the implementation of concurrent algorithms due to the direct mapping between patterns of communications and processes with language expressions; on one hand it becomes simpler since the language provides an abstraction for communication, on the other hand it changes the paradigm for implementing algorithms, since these new communication constructs have to be taken into account.

Moreover, concurrent programming adds a layer of abstraction over the parallel facilities of processors and operating systems, offering a high-level interface that allows the user to program modules of code to be executed in parallel threads [1].

Different languages offer different concurrency strategies depending on how they deal with shared state, that is, data structures that could be accessed from several processes. In this regard, there are two major fields (with some other variations):

- Actor-based concurrency [17] totally eliminates shared state by introducing a series of data structures called *actors* that store state and can mutate it locally.
- Process calculi or process algebra is a framework to describe systems that work with independent processes that interact between them using channels. One of the best known is called the *communicating sequential processes* (CSP) methodology [9], which is effectively stateless, with different processes reacting to a channel input without changing state, and writing to these channels. Unlike actor based concurrency, which keeps state local, in this case per-process state is totally eliminated, with all computation state managed as messages in a channel.
- Other, less well known models using, for instance, tuple spaces [7].

Most modern languages, however, follow the CSP abstraction, and it has become popular since it fits well other programming paradigms, like reactive and functional programming, and allows for a more efficient implementation, with less overhead, and with well-defined primitives. This is why we will use it in this paper for

*Produces the permission block, and copyright information
†The full version of the author's guide is available as `acmart.pdf` document

GECCO '19, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

creating new evolutionary algorithms that live *natively* in these environments and can thus be implemented easily in this kind of languages. We have chosen Perl 6, although other languages such as Go, are feasible alternatives.

In previous papers [22**?** , 23] we designed an evolutionary algorithm that fits well this architecture and explored its possibilities. That initial exploration showed that a critical factor within this algorithic model is communication between threads; therefore designing efficient messages is high-priority to obtain a good algorithmic performance and scaling. In this paper, we will test several communication strategoes: a lossless one that compresses the population, and a lossy one that sends a representation of population gene-wise statistics.

2 STATE OF THE ART

The problem of communication/synchronization between processes, which nowadays is accomplished by means of threads, has been the subject of long-standing study. One of the best efforts to formalize and simplify that matter is HoareâĂŹs *Communicating Sequential Processes* [9], that proposes an interaction description language which is the theoretical support for many libraries and modern programming languages. In such model, concurrent programs communicate on the basis of *channels*, a sort of pipe buffer used to interchange messages between the different processes or threads, either asynchronously or synchronously. Languages such as Go and Perl 6 implement this concurrency model (the latter including additional mechanisms such as *promises* or low-level access to the creation of threads).

The fact that messages have to be processed without secondary effects and that actors do not share any kind of state makes concurrent programming specially fit for functional languages or languages with functional features; this has made this paradigm specially popular for late cloud computing implementations; however, its presence in the EA world is not so widespread, although some efforts have lately revived the interest for this kind of paradigm [20]. Several years ago it was used in Genetic Programming [4, 10, 26] and recently in neuroevolution [18] but its occurrence in EA, despite being scarce in the previous years [8], has experimented a certain rise lately with papers such as [25] which perform program synthesis using the functional programming features of the ERlang language [3] for building an evolutionary multi-agent system.

Regarding functional languages, Erlang and Scala have embraced the actor model of concurrency and get excellent results in many application domains; Clojure is another one with concurrent features such as promises/futures, Software Transaction Memory and agents; Kotlin [19] has been recently used for implementing a functional evolutionary algorithm framework.

On the other hand, Perl 6 [**?**] uses different concurrency models, varying from implicit concurrency using a particular function that automatically parallelizes operations on iterable data structures, to explicit concurrency using threads. These both types of concurrency will be considered in this study whilst using the `Algorithm::Evolutionary::Simple`, a Perl 6 library introduced in last year's version of this very same conference [16].

Earlier efforts to study the issues of concurrency in EA are worth mentioning. For instance, the EvAg model [11] resorts to the underlying platform scheduler to manage the different threads of execution of the evolving agents; in this way the model scaled-up seamlessly to take full advantage of CPU cores. In the same avenue of measuring scalability, experiments were conducted in [13] comparing single and a dual-core processor concurrency achieving near linear speed-ups . The latter was later on extended in [14] by scaling up the experiment to up to 188 parallel machines, reporting speed-ups up to 960×, nearly four times the expected linear growth in the number of machines (when local concurrency were not taken into account). Other authors have addressed explicitly multi-core architectures, such as Tagawa [21] which used shared memory and a clever mechanism to avoid deadlock. Similarly, [12] used a message-based architecture developed in Erlang, separating GA populations as different processes, although all communication tooking place with a common central thread.

In previous papers [22, 24], we presented a proof of concept of the implementation of stateless evolutionary algorithms using Perl 6, based on a single channel model communicating threads for population evolving and mixing. In addition, we studied the effect of running parameters such as *generation gap* (similar to the concept of *time to migration* in parallel evolutionary algorithms) and population size, realizing that the choice of parameters may have a strong influence at the algorithmic level, but also at the implementation level, in fact affecting the actual wallclock performance of the EA.

3 EXPERIMENTAL SETUP

The baselinwe we are coming from is similar to the one used in previous experiments [22]. Our intention was to create a system that was not functionally equivalent to a sequential evolutionary algorithms, and that followed the principle of communicating sequential processes. In this kind of methodology, we will have processes (or threads) communicating state through channels. Every process itself will be stateless, reacting to the presence of messages in the channels it is listening to and sending result back to them, without changing state.

As in the previous papers, [23], we will use two groups of threads and two channels. We will see them in turns.

The two groups of threads perform the following functions:

- The *evolutionary* threads will be the ones that will be in principle running the evolutionary algorithm.
- The *mixing* threads will *mix* populations, and create new ones as a mixture of them.

The two channels carry messages that are equivalent to populations, but they do so in a different way:

- The *evolutionary* channel will be used for carrying non-evolved, or generated, populations.
- The *mixer* channel will carry, *in pairs*, evolved populations.

These will be connected as shown in Figure 1. The evolutionary group of threads will read only from the evolutionary channel, evolve for a number of generations, and place result in the mixer channel; the mixer group of threads will read only from the mixer channel, in pairs. From every pair, a random element is put back into the mixer channel, and a new population is generated and sent back to the evolutionary channel. The main objective of using

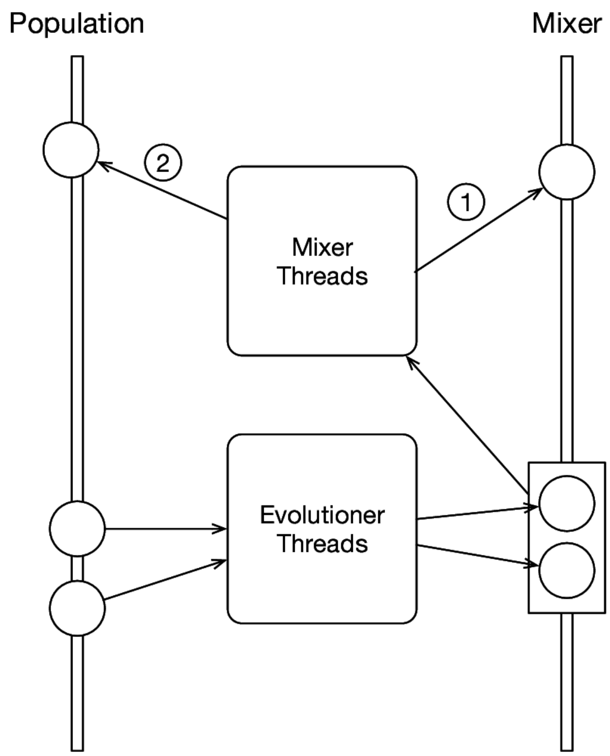

Figure 1: General scheme of operation of channels and thread groups.

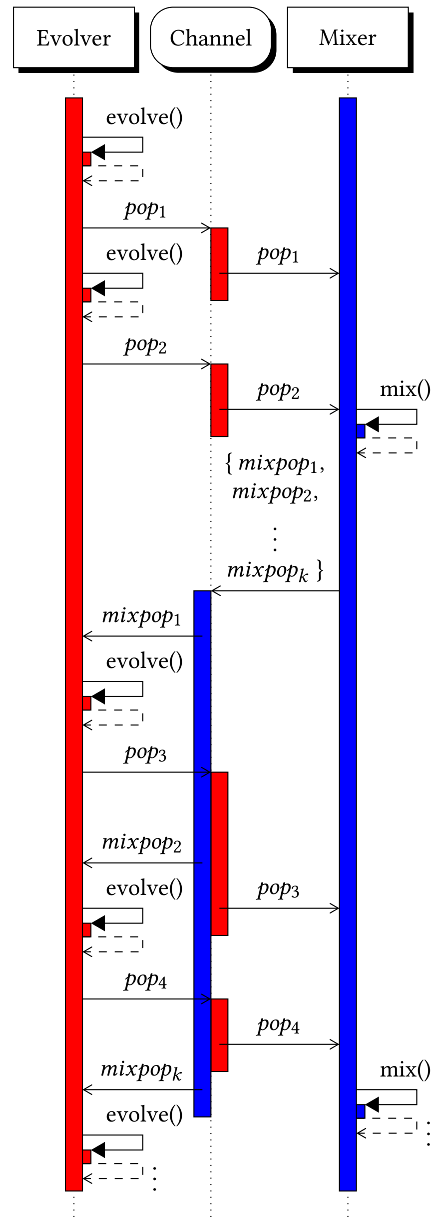

Figure 2: Schematic of communication between threads and channels for concurrent EAs. The two central bars represent the channel, and color corresponds to the *main* function they perform; blue for mixer, red for evolver. As the figure shows, the evolver threads always read from the mixer channel, and always write to the evolver channel.

two channels is to avoid deadlocks; the fact that one population is written always back to the mixer channel avoids starvation in the channel. How this runs in practice is shown in Figure 2, where the timeline of the interchange of messages between the evolver and mixer threads and evolver and mixer channels is clarified.

One of the problems of the baseline configuration was that communication took a great amount of time, adding some overhead to the algorithm. The *message* consisted of the whole population, and the size increased with population size, obviously. This tipped the balance between communication and computation towards communication, so that the more threads, the more communication was taking place. Our first intention in this paper was to slim down messages so that they took less bandwith (or memory) and less time to send and process. In general, this strategy also can be framed in the context of migration strategies, since that is the most similar thing in the context of parallel algorithms. In parallel algorithms, an adequate selection of migration strategies, balancing exploration and exploitation, is the key to achieving high performance, as indicated in [5]. In the context of concurrent evolutionary algorithms we will talk about *population messages*, but their effect is going to be similar. For this paper, we introduced two different messaging strategies:

- One we have called *EDA*, or estimation of distribution algorithm, whose basic idea is that the population message will contain the probability distribution over each gene. In this sense, this strategy is similar to the one presented by de la Ossa et al. in [6]. Not being an estimation of distribution algorithm *per se*, since the evolutionary thread runs a canonical genetic algorith, when the message is being composed, every (binary) gene of the 25% best individuals in the population is examined, and an array with the probabilities for each gene is sent to the mixer thread. The *mixer* thread, in turn, just takes randomly one probability from each of the two *populations* (actually, distributions), instead of working on individuals. While in the baseline strategy the selection took place in the mixer thread, that eliminated half the population, in this case the selection takes place when composing the message, since just the 25% best individuals are selected to compute the probability distribution of genes. When the evolver thread reads the message, it rebuilds the population based on this distribution.

- The second is called *compress*, and it simply bit-packs the population, without the fitness, into a message which uses 1 bit per individual, and then 64 bits, or simply 8 bytes, to transmit a single individual in the population. This strategy is equivalent to the baseline, except it introduces an additional step of evaluating the population when mixing and receiving it from the evolutionary channel. It is hoped that this additional evaluation overhead does compensates the communication overhead that is eliminated.

What we want to find out in these set of experiments is what is the generation gap that gives the best performance in terms of raw time to find a solution, as well as the best number of evaluations per second. In order to do that, we prepared an experiment using the OneMax function with 64 bits, a concurrent evolutionary algorithm such as the one described in [22], which is based in the free Perl 6 evolutionary algorithm library `Algorithm::Evolutionary::Simple`, and run the experiments in a machine with Ubuntu 18.04, an AMD Ryzen 7 2700X Eight-Core Processor at 2195MHz. The Rakudo version was 6.d, which had recently been released with many improvements to the concurrent core of the language. All scripts, as well as processing scripts and data obtained in the experiments are available, under a free license, from our GitHub repository.

We used a population size of 256, as well as generation gaps increasing from 8 to 64. Many experiments were run for every configuration, up to 150 in some cases. We logged the upper bound of the number of evaluations needed (by multiplying the number of messages by the number of generations and number of individuals evaluated; this means that this number will be an upper bound, and not the exact number of evaluations until a solution is reached). We will first look at the general picture by plotting the wallclock time in seconds (measured by taking the time of the starting of the algorithm and the last message and subtracting the latter from the former) vs the number of evaluations that have been performed. The result is shown in Figure **??**. Experiments with different generation gaps are shown with different colors (where available) and shapes, and they spread in an angle which is roughly bracketed by the experiments with a generation gap of 8, which need the most time for the same number of evaluations, and the experiments with a gap of 16, which usually need the least. The experiments with gaps = 32 or 64 are somewhere in between.

In that same chart it can also be observed that the number of evaluations needed to find the 64 bit OneMax solution is quite different. We make a boxplot of the number of evaluations vs the generation gap in Figure **??**. This figure shows an increasing number of evaluations per gap size. Differences are significant between every generation gap and the next. This increasing number of evaluations per generation gap is probably due to the fact that the increasing number of isolated generations makes the population lose diversity, making finding the solution increasingly difficult. This is the same effect observed in parallel algorithms, as reported in [5], so it is not unexpected. What is unexpected is the combination of generation gap size and the concurrent algorithm, since it is impossible to know in advance what is the optimal computation to communication balance.

We plot the number of evaluations per second in Figure **??**. These show a big difference for a generation gap of 16, with a number of evaluations which is almost 50% higher than for the rest of the generation gaps, where the difference is not so high.

The number of evaluation per second does not follow a clear trend. It falls and remains flat for a generation gap higher than 16; it is also slightly higher than for the minimum generation gap that has been evaluated, 8. This generation gap, however, presents also the lowest number of evaluations to solution, which means that, on average, the solution will be found faster with a generation gap of 8 or 16. This is shown in Figure **??**.

4 CONCLUSIONS

In this paper we have set out to explore the interaction between the generation gap and the algorithmic parameters in a concurrent and stateless evolutionary algorithm. From the point of view of the algorithm, increasing the generation gap favors exploitation over exploration, which might be a plus in some problems, but also decreases diversity, which might lead to premature convergence; in a parallel setting, this will make the algorithm need more evaluations to find a solution. The effect in a concurrent program goes in the opposite direction: by decreasing communication, the amount of code that can be executed concurrently increases, increasing performance. Since the two effects cancel out, in this paper we have used a experimental methodology to find out what is the combination that is able to minimize wallclock time, which is eventually what we are interested in by maximizing the number of evaluations per second while, at the same time, increasing by a small quantity the number of evaluations needed to find the solution.

For the specific problem we have used in this short paper, a 64-bit onemax, the generation gap that is in that area is 16. The time to communication for that specific generation gap is around 2 seconds, since 16 generations imply 4096 evaluations and evaluation speed is approximately 2K/s. This gives us a ballpark of the amount of computation that is needed for concurrency to be efficient. In this case, we are sending the whole population to the communication channel, and this implies a certain overhead in reading, transmitting and writing. Increasing the population size also increases that overhead.

We can thus deduce than the amount of computation, for this particular machine, should be on the order of 2 seconds, so that it effectively overcomes the amount of communication needed. This amount could be played out in different way, for instance by increasing the population; if the evaluation function takes more time, different combinations should be tested so that no message is sent unless that particular amount of time is reached.

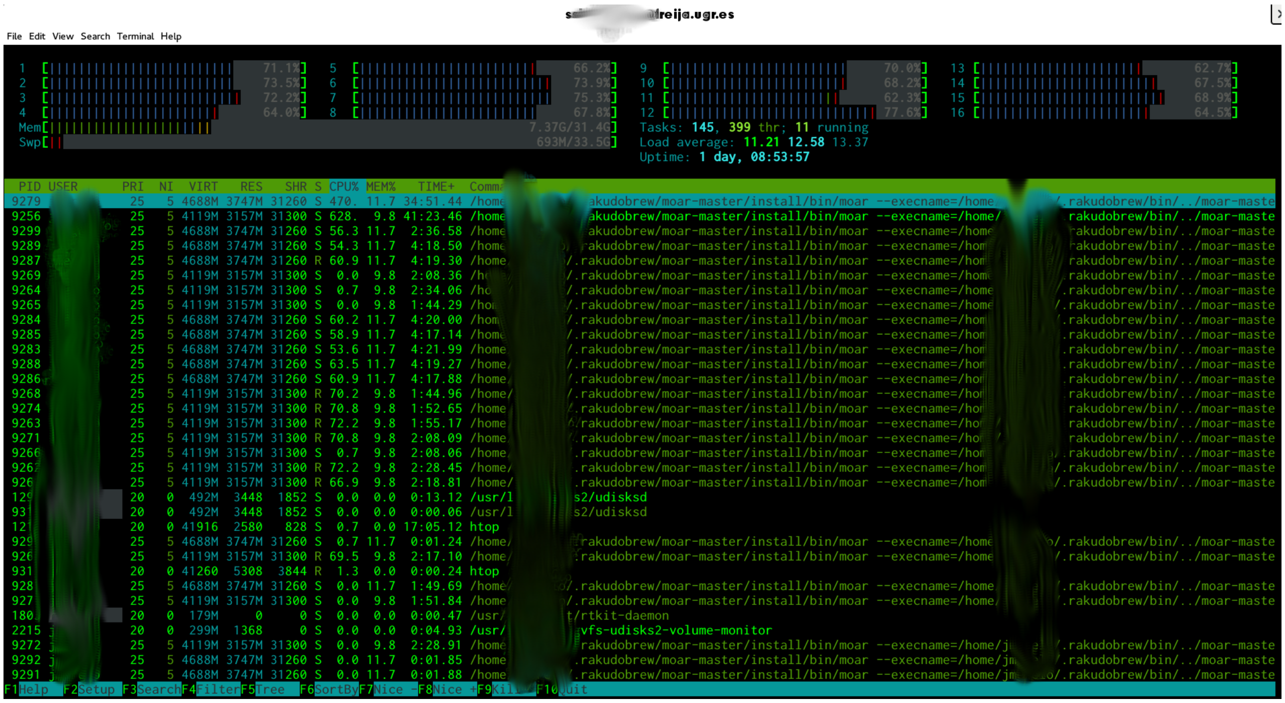

Figure 3: Screenshot using the htop utility of the used machine running two experiments at the same time. As it can be seen, all processors are kept busy, with a very high load average.

With these conclusions in mind, we can set out to work with other parameters, such as population size or number of initial populations, so that the loss of diversity for bigger population sizes is overcome. Also we have to somehow overcome the problem of the message size by using a statistical distribution of the population, or simply other different setup. This is left as future work.

ACKNOWLEDGMENTS

Acknowledgements taking
this much
space

REFERENCES

[1] Gregory R Andrews. 1991. *Concurrent programming: principles and practice.* Benjamin/Cummings Publishing Company San Francisco.

[2] Joe Armstrong. 2003. Concurrency Oriented Programming in Erlang. (2003). http://ll2.ai.mit.edu/talks/armstrong.pdf

[3] Adam D Barwell, Christopher Brown, Kevin Hammond, Wojciech Turek, and Aleksander Byrski. 2017. Using program shaping and algorithmic skeletons to parallelise an evolutionary multi-agent system in Erlang. *Computing and Informatics* 35, 4 (2017), 792–818.

[4] Forrest Briggs and Melissa O'Neill. 2008. Functional genetic programming and exhaustive program search with combinator expressions. *Int. J. Know.-Based Intell. Eng. Syst.* 12, 1 (Jan. 2008), 47–68. http://dl.acm.org/citation.cfm?id=1375341.1375345

[5] Erick Cantú-Paz. 1999. Migration Policies and Takeover Times in Genetic Algorithms. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1 (GECCO'99).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 775–775. http://dl.acm.org/citation.cfm?id=2933923.2934003

[6] Luis delaOssa, José A. Gámez, and José M. Puerta. 2004. Migration of Probability Models Instead of Individuals: An Alternative When Applying the Island Model to EDAs. In *Parallel Problem Solving from Nature - PPSN VIII*, Xin Yao, Edmund K. Burke, José A. Lozano, Jim Smith, Juan Julián Merelo-Guervós, John A. Bullinaria, Jonathan E. Rowe, Peter Tiňo, Ata Kabán, and Hans-Paul Schwefel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 242–252.

[7] David Gelernter. 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 1 (1985), 80–112.

[8] John Hawkins and Ali Abdallah. 2001. A Generic Functional Genetic Algorithm. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications (AICCSA '01).* IEEE Computer Society, Washington, DC, USA, 11–. http://dl.acm.org/citation.cfm?id=872017.872197

[9] C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. https://doi.org/10.1145/359576.359585

[10] Lorenz Huelsbergen. 1996. Toward simulated evolution of machine-language iteration. In *Proceedings of the First Annual Conference on Genetic Programming (GECCO '96).* MIT Press, Cambridge, MA, USA, 315–320. http://dl.acm.org/citation.cfm?id=1595536.1595579

[11] Juan-Luis Jiménez-Laredo, A. E. Eiben, Maarten van Steen, and Juan-Julián Merelo-Guervós. 2010. EvAg: a scalable peer-to-peer evolutionary algorithm. *Genetic Programming and Evolvable Machines* 11, 2 (2010), 227–246.

[12] Kittisak Kerdprasop and Nittaya Kerdprasop. 2012. Concurrent Computation for Genetic Algorithms. In *Proceedings of the 1st International Conference on Software Technology.* 79–84.

[13] J.L.J. Laredo, P.A. Castillo, A.M. Mora, and J.J. Merelo. 2008. Exploring Population Structures for Locally Concurrent and Massively Parallel Evolutionary Algorithms. In *WCCI 2008 Proceedings.* IEEE Press, 2610–2617. http://atc.ugr.es/I+D+i/congresos/2008/CEC_2008_2610.pdf

[14] Juan Luis Jiménez Laredo, Pascal Bouvry, Sanaz Mostaghim, and Juan Julián Merelo Guervós. 2012. Validating a Peer-to-Peer Evolutionary Algorithm. In *Applications of Evolutionary Computation - EvoApplications 2012, Málaga, Spain, April 11-13, 2012, Proceedings.* 436–445. https://doi.org/10.1007/978-3-642-29178-4_44

[15] Xia Li, Kunqi Liu, Lixiao Ma, and Huanzhe Li. 2010. A Concurrent-Hybrid Evolutionary Algorithms with Multi-child Differential Evolution and Guotao Algorithm Based on Cultural Algorithm Framework. In *Advances in Computation and Intelligence*, Zhihua Cai, Chengyu Hu, Zhuo Kang, and Yong Liu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 123–133.

[16] Juan Julián Merelo Guervós and José Mario García Valdez. 2018. Performance improvements of evolutionary algorithms in perl 6. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2018, Kyoto, Japan, July 15-19, 2018*, Hernán E. Aguirre and Keiki Takadama (Eds.). ACM, 1371–1378. https://doi.org/10.1145/3205651.3208273

[17] Hans Schippers, Tom Van Cutsem, Stefan Marr, Michael Haupt, and Robert Hirschfeld. 2009. Towards an actor-based concurrent machine model. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 4–9.

[18] Gene I. Sher. 2013. *Handbook of Neuroevolution Through Erlang*. Springer.

[19] Jed Simson and Michael Mayo. 2017. Open-Source Linear Genetic Programming. (2017).

[20] Jerry Swan, Steven Adriaensen, Mohamed Bishr, Edmund K Burke, John A Clark, Patrick De Causmaecker, Juanjo Durillo, Kevin Hammond, Emma Hart, Colin G Johnson, et al. 2015. A research agenda for metaheuristic standardization. In *Proceedings of the XI Metaheuristics International Conference*.

[21] K. Tagawa. 2012. Concurrent differential evolution based on generational model for multi-core CPUs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7673 LNCS (2012), 12–21. cited By (since 1996) 0.

[22] Same A. U. Thors. 2018. Some title. In *Proceedings*. ACM.

[23] Same A. U. Thors. 2018. Some title. In *Proceedings*. Springer.

[24] Same A. U. Thors. 2018. Some title. In *Proceedings*. ACM.

[25] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. 2018. Synthesis of Differentiable Functional Programs for Lifelong Learning. *arXiv preprint arXiv:1804.00218* (2018).

[26] Paul Walsh. 1999. A Functional Style and Fitness Evaluation Scheme for Inducting High Level Programs. In *Proceedings of the Genetic and Evolutionary Computation Conference*, Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith (Eds.), Vol. 2. Morgan Kaufmann, Orlando, Florida, USA, 1211–1216. http://www.cs.bham.ac.uk/~wbl/biblio/gecco1999/GP-455.ps