Mapping evolutionary algorithms to a reactive, stateless architecture

Using a modern concurrent language

Juan J. Merelo
Universidad de Granada
Granada, Spain
jmerelo@ugr.es

José-Mario García-Valdez
Instituto Tecnológico de Tijuana
Tijuana, Mexico
mario@tectijuana.edu.mx

ABSTRACT

CCS CONCEPTS

• **Theory of computation → Evolutionary algorithms**; • **Computing methodologies** → *Distributed algorithms*;

KEYWORDS

Microservices, distributed computing, event-based systems, Kappa architecture, stateless algorithms, algorithm implementation, performance evaluation, distributed computing, pool-based systems, heterogeneous distributed systems, serverless computing, functions as a service.

ACM Reference Format:
Juan J. Merelo and José-Mario García-Valdez. 2018. Mapping evolutionary algorithms to a reactive, stateless architecture: Using a modern concurrent language. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3205651.3208317

1 INTRODUCTION

Genetic algorithms (GA) [8] are currently one of the most widely used meta-heuristics to solve engineering problems. Furthermore, parallel genetic algorithms (pGAs) are useful to find solutions of complex optimizations problems in adequate times [16]; in particular, problems with complex fitness. Some authors [1] state that using pGAs improves the quality of solutions in terms of the number of evaluations needed to find one. This reason, together with the improvement in evaluation time brought by the simultaneous running in several nodes, have made parallel and distributed evolutionary algorithms a popular methodology.

Implementing evolutionary algorithms in parallel is relatively straightforward, but programming paradigms used for the implementation of such algorithms is far from being an object of study. Object oriented or procedural languages like Java and C/C++ are mostly used. Even when some researchers show that implementation matters [18], parallels approaches in new languages/paradigms is not normally seen as a land for scientific improvements.

New parallel platforms have been identified as new trends in pGAs [16], however only hardware is considered. Software platforms, specifically programming languages, remain poorly explored; only Ada [19] and Erlang [5, 13] were slightly tested.

The multicore's challenge [10] shows a current need for making parallel even the simplest program. But this way leads us to use and create design patterns for concurrent algorithms; the conversion of a pattern into a language feature is a common practice in the programming languages domain, and sometimes that means a language modification, others the creation of a new one.

This work explores the advantages of Perl6 [21], a relatively new and decidedly non mainstream languages, since it is not included in the top ten of any most popular languages ranking) with concurrent and functional features in order to develop EAs in its parallel versions through concurrency. This paper, as well as similar ones preceding it [3, 7], is motivated by the lack of community attention on the subject and the belief that using concepts that simplify the modeling and implementation of such algorithms might promote their use in research and in practice.

This research is intended to show some possible areas of improvement on architecture and engineering best practices for concurrent-functional paradigms, as was made for Object Oriented Programming languages [17], by focusing on pGAs as a domain of application and describing how their principal traits can be modeled by means of concurrent-functional languages constructs. We are continuing the research reported in [2, 7].

The rest of the paper is organized as follows. Next section presents the state of the art in concurrent and functional programming language paradigms and its potential use for implementing pGAs. We present two different versions of a concurrent evolutionary algorithm in Section 3, to be followed by actual results in section 4. Finally, we draw the conclusions and present future lines of work in section 5.

2 STATE OF THE ART

Developing correct software quickly and efficiently is a never ending goal in the software industry. Novel solutions that try to make a difference providing new abstraction tools outside the mainstream of programming languages have been proposed to pursue this goal; two of the most promising are the functional and the concurrent.

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208317

The concurrent programming paradigm (or concurrency oriented programming [4]) is characterized by the presence of programming constructs for managing processes like first class objects. That is, with operators for acting upon them and the possibility of using them like parameters or function's result values. This simplifies the coding of concurrent algorithms due to the direct mapping between patterns of communications and processes with language expressions.

Concurrent programming is hard for many reasons, the communication/synchronization between processes is key in the design of such algorithms. One of the best efforts to formalize and simplify that is the Hoare's *Communicating Sequential Processes* [11], this interaction description language is the theoretical support for many libraries and new programming languages.

When a concurrent programming language is used normally it has a particular way of handling units of execution, being independent of the operation system has several advantages: one program in those languages will work the same way on different operating systems. Also they can efficiently manage a lot of processes even on a mono-processor machine.

Functional programming paradigm, despite its advantages, does not have many followers. Several years ago was used in Genetic Programming [6, 12, 23] and recently in neuroevolution [20] but in GA its presence is practically nonexistent [9].

This paradigm is characterized by the use of functions as first class concepts, and for discouraging the use of state changes, with functions mapping directly input to output without having any side effect. The latter is particularly useful for develop concurrent algorithms in which the communication by state changes is the origin of errors and complexity. Also, functional features like closures and first class functions in general, allow to express in one expression patterns like *observer* which in language like Java need so many lines and files of source code.

The field of programming languages research is very active in the Computer Science discipline. To find software construction tools with new and better means of algorithms expression is welcome. In the last few years the functional and concurrent paradigms have produced a rich mix in which concepts of the first one had been simplified by the use of the second ones.

Among this new generation, the languages Erlang and Scala have embraced the actor model of concurrency and get excelentes results in many application domains; Clojure is another one with concurrent features such as promises/futures, Software Transaction Memory and agents. All of these tools have processes like built-in types and scale beyond the restrictions of the number of OS-threads. On the other hand, Perl 6 [22] uses different concurrency models, that go 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 analyzed in this paper.

3 CONCURRENT EVOLUTIONARY ALGORITHMS AND ITS IMPLEMENTATION

The implementation of evolutionary algorithms in a concurrent environment must have several features:

- They must be *reactive*, that is, functions respond to events, and not procedural or sequential.
- Functions responding to events are also first class objects and are stateless, having no secondary effects. These functions have to be reentrant, that is, with the capability of being run in a thread without exclusion of other functions.
- Functions communicate with each other exclusively via channels, which can hold objects of any kind but are not cached or buffered. Channels can be shared, but every object can be read from a channel only once.

In general, an evolutionary algorithm consists of an interative procedure where, after generating an initial set of individuals, these individuals are evaluated, and then they reproduce, with errors and combination of their features, with a probability that is proportional to their fitness. As long as there is variation and survival of the fittest, an evolutionary algorithm will work. However, the usual way of doing this is through a series of nested loops, with possibly asynchronous operation in a parallel context when communicating with other *islands* or isolated populations. However, the concept of loop itself implies state, in the shape of the generation counter, or even with the population itself that is handled from one iteration step to the next one.

Getting rid of these states, however, leads to many different algorithms which are not functionally equivalent to the canonical genetic algorithm above. Of course, a functional equivalent is also possible in this environment, with non-terminating *islands* running every one of them on a different thread, and communicating via channels. Although this version is guaranteed to succeed, we are looking for different implementations that, while keeping the spirit of the evolutionary algorithm, maps themselves better to a multi-threaded architecture and a concurent language such as Go, Scala or Perl 6.

This is why in this paper we are going to examine two different architectures, which basically differ in the granularity with which they perform the evolutionary algorithm.

3.1 Individual-level concurrency

In this version of the algorithm, all functions operate on single individuals or sets of them. We are going to use three different channels:

- Channel individual, which contains chromosomes without a fitness function. A subchannel of this channel takes the chromosomes in groups.
- Channel evaluated, which contains chromosomes paired with their fitness function. This channel receives individuals one by one, but emits them in groups of three.
- Channel output, which is used for logging what is happening in the other two channels and printing output at the end of the experiment.

There are two functions feeding these channels.

- Evaluator reacts to the individual channel, picking and evaluating a single individual and emits it to the evaluated as well as output channel as an object that contains the original chromosome and the computed fitness.
- Reproducer picks three individuals from the evaluated channel, generates a new couple using crossover, and emits it to

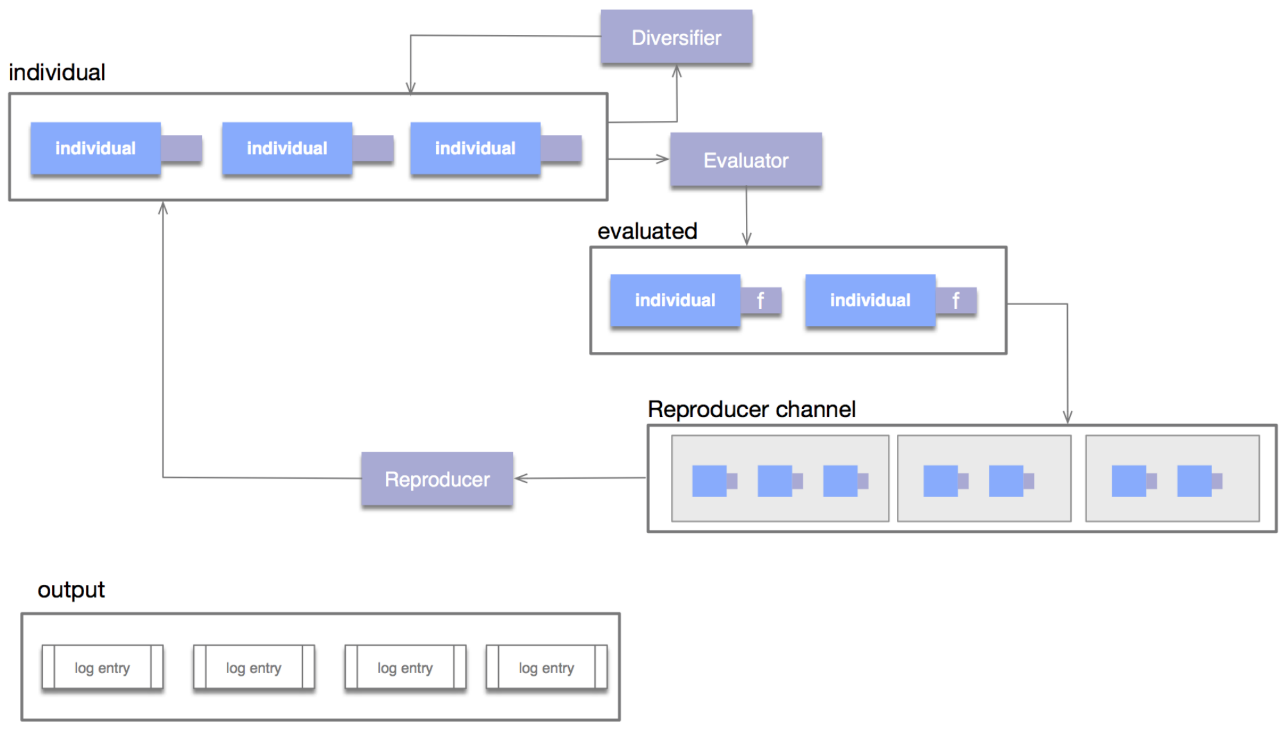

Figure 1: Channels and functions used in the individual-level concurrency version of the algorithm.

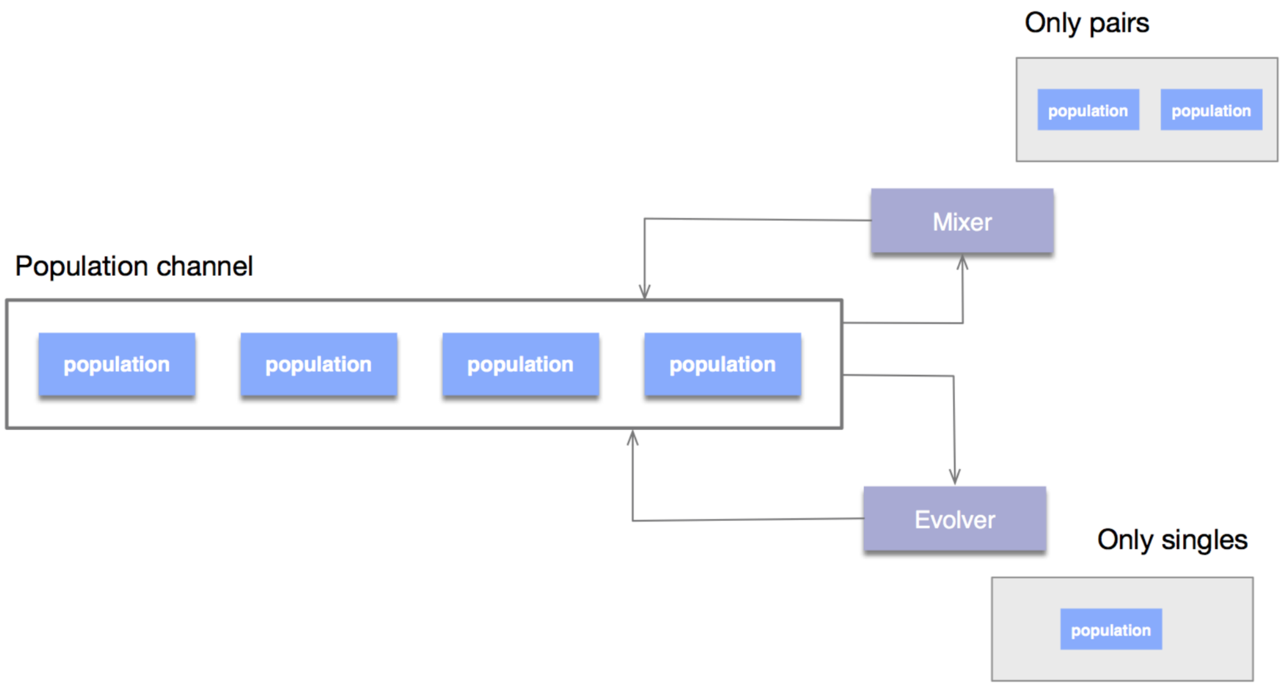

Figure 2: Channel and functions used in the population-level concurrency version of the algorithm.

the individual channel. This function also acts as selector, and in fact it is similar to 3 tournament, since it takes three individuals and returns only two of them to the channel, along with the two individuals that have been generated via crossover and mutation.

- Diversifier is a re-broadcasting of the individual channel, picks a group of individuals and shuffles it, putting it back into the same channel, giving them a different order in the buffer.

How channels and functions relate to and communicate with each other is represented in Figure 1. The functions described above rebroadcast the values they read from the cannel when needed to other channels so that all channels are kept fed and a deadlock situation is not produced. This could happen, for instance, if the reproducer channel, which takes individuals in pairs, is only able to read a single one; since it is waiting for a second individual it is not producing new ones and the algorithm will stall. This could be fixed in a different way by changing from a reactive architecture to a *polling* architecture, but that kind of architecture also introduces overhead by polling when it is not needed. You have to balance when designing these types of algorithms, anyway; polling is another possibility, but one we are not exploring in this paper.

The concurrency of this situation implies that we can run as many copies as available of every one of them. Also that there is an initial process where you generate the initial population, a series of individuals which must be even, and bigger than the number of individuals used in the diversifier. This is equivalent to an initial population, although in this case there is no real *population*, since individuals are considered in groups of three.

Depending on the overhead emission and reception adds, it is possible that the performance of this channel is not the adequate one, even if theoretically it is sound. That is why we have also proposed next a coarse-grained version where the function process whole populations.

3.2 Population-level concurrency

In this case, the algorithm uses a single channel that emits and receives populations. However, this channel is also re-broadcast as another channel that takes the population in pairs. Having a single channel, even is with different threads, will make several threads concurrently process populations that will evolve in complete independence. This is why there are two functions:

- Singles takes single populations and evolves them for a number of generations. It stops if it finds the solution, and closes the channel at the same time.
- Pairs reads pairs of populations from the sub-channel and mixes them, creating a new population with the best members of both populations. This *mixer* is equivalent to a process of migration that takes members from one population to another. Since this function takes two elements from the channel, it must leave two elements in the channel too. What it does is it emits back a randomly chosen population in the pair.

Additionally, there must be a function, which can be concurrent, to create the initial population. The process of migration performed by the mixer is needed to overcome the *stateless* nature of the concurrent process. The state is totally contained in the population; the mixer respects this state of affairs by using only this information to perform the evolutionary algorithm.

This algorithm has several parameters to tune:

- **Number of generations** that every function runs. This parameter is equivalente to the time needed to perform some kind of migration, since it is the time after which populations are sent back to the channel for mixing and further evolution.
- **initial populations** The channel must never be empty, so some initial random populations must be generated, always in pairs.

3.3 Notes on implementation using Perl 6

Perl 6 [22] has been chosen to perform the implementation of these two different versions of a concurrent evolutionary algorithm. This choice has been due mainly to the existence of an open source evolutionary algorithm library, recently released by the authors and called `Algorithm::Evolutionary::Simple`. This library, released to the repository of common Perl 6 modules and called CPAN, includes functions for the implementation of a very simple evolutionary algorithm for optimizing onemax, Royal Road or any other benchmark function.

Perl 6 [14] is, despite its name, a language that is completely different from Perl, designed for scratch to implement most modern language features: meta-object protocols, concurrency, and functional programming. It does not have a formal grammar, but is rather defined by the tests a compiler or interpreter must pass in order to be called “Perl 6”. It consists of a virtual machine and a just in time compiler which is written mostly in Perl 6 itself, so that it can be easily ported from one virtual machine to others. Although it can target many different virtual machines, the current “official” implementation includes a virtual machine called MoarVM and a compiler called Rakudo. All together they compose the so-called *Rakudo start* distribution, a *stable* distribution of compiler + virtual machine that is released every 4 monts from GitHub and to package repositories.

The advantage of using Perl 6 is that it combines the expressivity of an interpreted language with the power of concurrency. Not very many languages nowadays include concurrency as a base feature; Go, Scala and Erlang are some of them. The concurrency in Go is done in a similar way to Perl 6, using channels, but Go is a compiled, non-functional language.

The main disadvantage of Perl 6 is raw performance, which is much slower than Go, although in general, similar although slower than other interpreted languages such as Python or Perl. Language performance is not an static feature, and it usually improves with time; in a separate paper, we have proved how speed has increased by orders of magnitude since it was released a few years ago.

This paper, however, is focused on the algorithmic performance more than the raw performance, so suffice it to say that Perl 6 performance was adequate for running these experiments in a reasonable amount of time.

The module used, as well as the code for the experiments, is available under a free license.

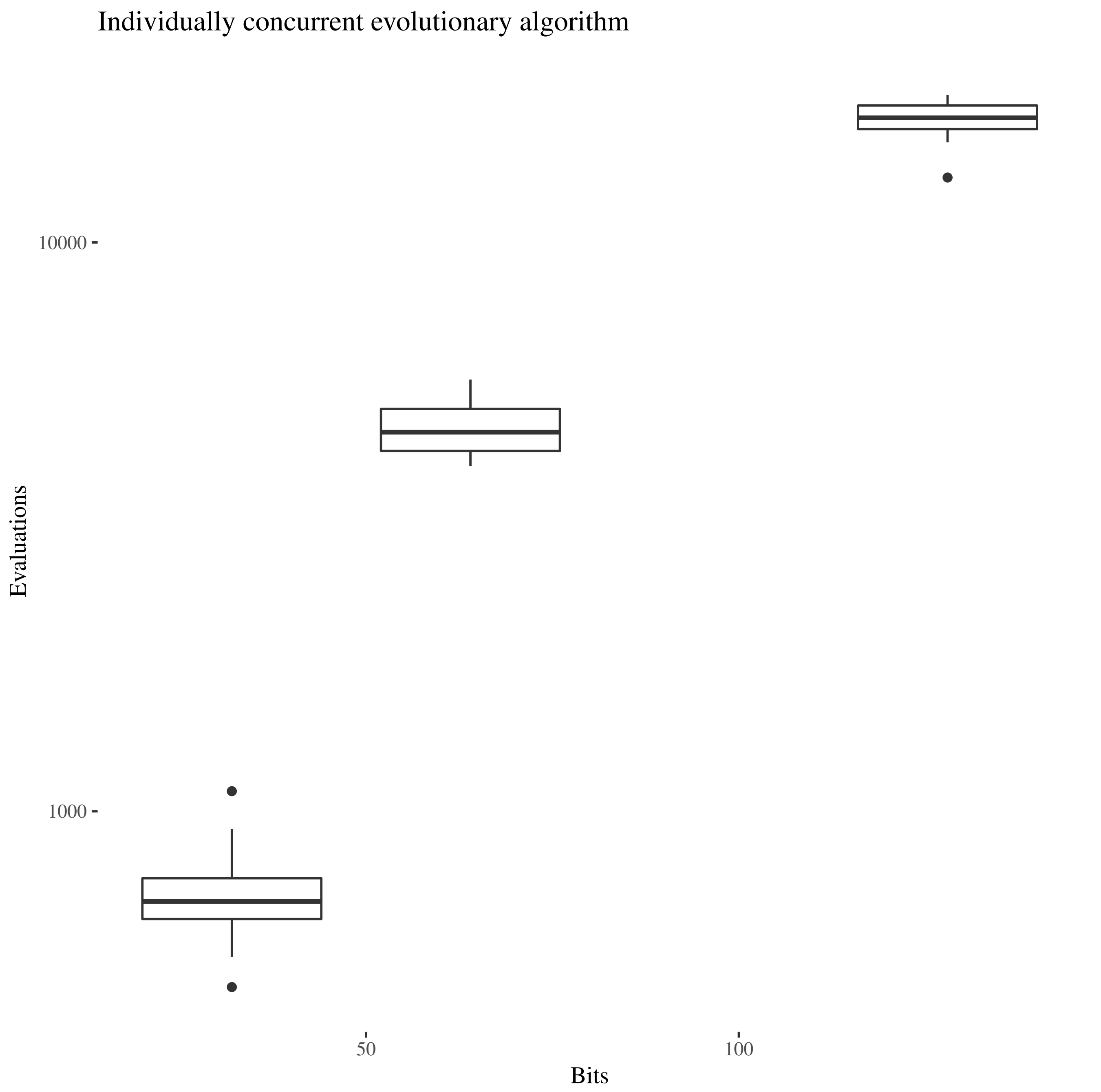

Figure 3: Boxplot of the number of evaluations needed for different number of bits in the maxones problem. Please note that axes x and y both have a logarithmic scale.

4 EXPERIMENTAL SETUP AND RESULTS

In order to perform the experiments, we used Linux boxes (with Ubuntu 14.04 and 16.04), the latest version of the Perl 6 compiler and virtual machine. First we used a selecto-recombinative evolutionary algorithm, with no mutation, in order to find out what's the correct population for every problem size [15]. This method sizes populations looking for the minimal size that achieves a 95% success rate on a particular problem and problem size; in this case, size 512 was the ideal for the maxones problem with size 64. This size was used as a base for the rest of the problem sizes; since the real evolutionary algorithm actually uses mutation, the population was halved for the actual experiments. This population size is more dependent on the problem itself than on the particular implementation, that is why we use it for all implementations. First we run

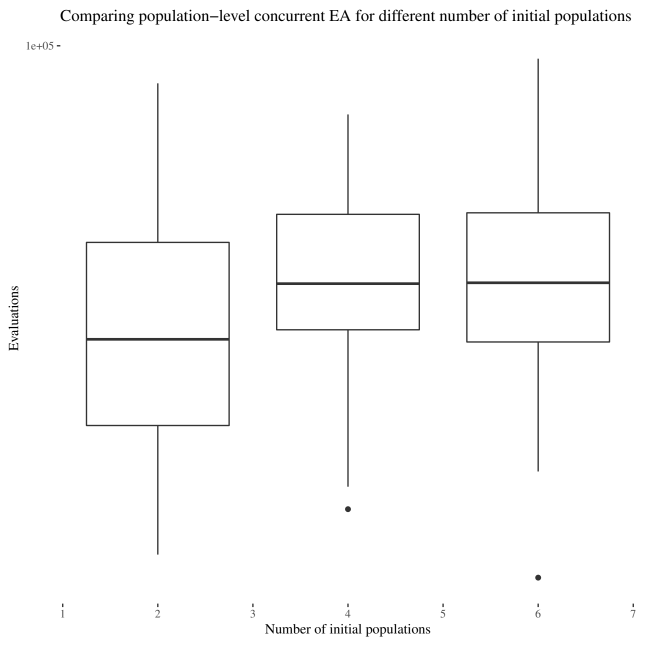

Figure 4: Boxplot comparing the number of evaluations needed for solving the 64 bit onemax problem using the population-leven concurrent algorithm with different number of initial populations.

a basic evolutionary algorithm with different chromosome sizes, to get a baseline of the number of evaluations needed for finding the solution in that case. Time needed was the main requisite for choosing the different sizes, although we think that scaling should follow more or less the same trend as shown for smaller sizes. We compared mainly the number of evaluations needed, since that is the main measure of the quality of the algorithm.

We show in Figure 3 the logarithmic chart of the number of evaluations that are reached for different, logarithmically growing, chromosome sizes using the individually concurrent evolutionary algorithm. There is a logical increase in the number of evaluations needed, but the fact that it is a low number and its scaling prove that this simple concurrent implementation is indeed an evolutionary algorithm, and does not get stuck in diversity traps that take it to local minimum. The number of evaluations is, in fact, quite stable.

We did the same for the population-level concurrent algorithm; however, since this one has got parameters to tune, we had to find a good result. In order to do that, we tested different number of initial populations placed in the channel, since this seems to be the critical parameter, more than the number of generations until mixing. The results are shown in Figure 4. The difference between using 4 and 6 initial populations is virtually none, but there is a slight advantage if you use only 2 initial populations to kickstart the channel. Please bear in mind that, in this case, the concept of *population* is slightly different from the one used in island EAs. While in the latter the population does not move from the island, in this case populations are read from channels and acted upon, and in principle there could be as many initial or unread populations as wanted or needed; every function in every thread will process a single population nonetheless.

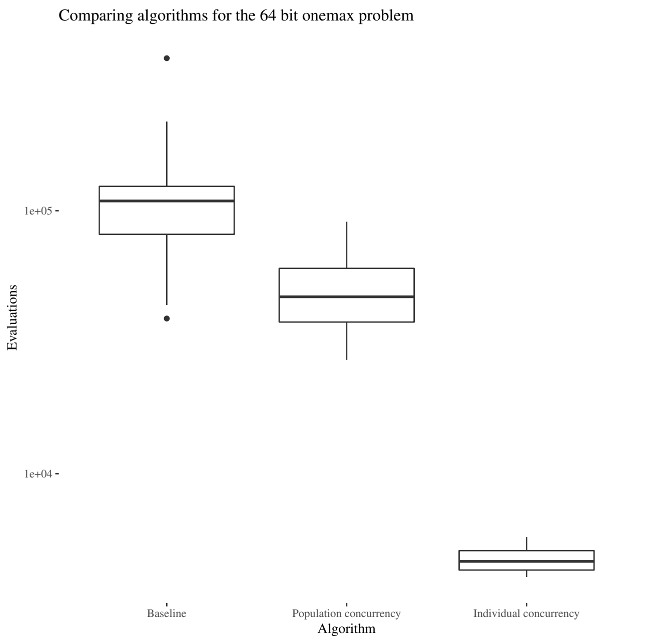

Figure 5: Boxplot comparing the number of evaluations needed for solving the 64 bit onemax problem in the individually concurrent and canonical EA version.

We can now compare these two algorithms with the baseline EA. This is a canonical evolutionary algorithm using bitflip mutation and two-point crossover, using roulette wheel as a selection method. It has, as the rest of the implementations of the algorithms, been implemented using `Algorithm::Evolutionary::Simple`, the free software module in Perl 6.

The comparison is shown in Figure 5, which shows, in a logarithmic y scale, a boxplot of the number of evaluations for the baseline, as well as the two different concurrent algorithms, at the population and individual level. As it can be seen, this last algorithm outperforms the other two, achieving the same result using many less evaluations, almost one order of magnitude less. In fact, both concurrent algorithms are better than the baseline, and please note this measures the number of evaluations, equivalent to the algorithmic complexity, and not time. This figure is the base to reach our conclusions.

5 CONCLUSIONS AND DISCUSSION

It is natural to take advantage of the multithreading and multiprocess capabilities of modern architectures to make evolutionary or other population-based algorithms faster; that can be done in a very straightforward way by parallelizing the evolutionary algorithm using the many available models, such as island model; however, it is possible that adapting the algorithm itself to the architecture makes its performance better.

However, this change implies also a different vision of the algorithm, that is why the first thing that has to be evaluated is the actual number of evaluations that need to be done to solve the problem. This is what we have done in this paper. We have proposed two different concurrent implemtations of an evolutionary algorithms with different *grain*: a *fine-grained* one that acts at the individual level, and a *coarse-grained* one that acts at the level of populations.

The individual-level concurrent EA shows a good scaling across problem size; besides, when comparing it with the population-level concurrent EA and the canonical and sequential evolutionary algorithm, it obtains a much better performance, being able to obtain the solution with a much lower evaluation budget. Second best is the population-level concurrent algorithm, to be followed by the baseline canonical GA, which obtains the worse result. This proves that, even from the purely algorithmic point of view, concurrent evolutionary algorithms are better than sequential algorithms. If we consider time, the difference increases, since the only sequential part of the concurrent algorithms is reading from the channels, but once reading has been done the rest of the operations can be performed concurrently, not to mention every function can have as many copies as needed running in different threads.

These results are not so much inherent to the concurrency itself as dependent on the selection operators that have been included in the concurrent version of the algorithms. The selection pressure of the canonical algorithm is relatively low, depending on roulete wheel. The population-level concurrent algorithm eliminates half the population with the worst fitness, although every generation it is running a canonical GA identical to the baseline; however, this exerts a high selective pressure on the population which, combined with the increased diversity of running two populations in parallel, results in better results. Same happens with the individual-level concurrent EA: the worst of three is always eliminated, which exerts a big pressure on the population, which thus is able to find the solution much faster. Nothing prevents us from using these same mechanisms in an evolutionary algorithm, which would then be functionally equivalent to these concurrent algorithms, but we wanted to compare a canonical EA to *canonical* concurrent evolutionary algorithms, at the same time we compare different versions of them; in this sense, it is better to use this individual-level concurrent algorithm in future versions of the evolutionary algorithm library.

The main conclusion of this paper is that evolutionary algorithms can benefit from concurrent implementations, and that these should be as fine grained as possible. However, a lot of work remains to be done. One line of research will be to try and use the implicitly concurrent capabilities of Perl 6 to perform multi-threaded evaluation or any other part of the algorithm, which would delegate the use of the threading facilities to the compiler and virtual machine. That will have no implications on the number of evaluations, but will help make the overall application faster.

Of course, time comparisons will also have to be made, as well as a more thorough exploration of the parameter space of the population-level evolutionary algorithm. Since this type of algorithm has a lower overhead, communicating via channels with lower frequency, it could be faster than the individual-level concurrent EA. Measuring the scaling with the number of thread is also an interesting line to pursue; since our architecture is using single channels, this might eventually be a bottleneck, and will prevent scaling to an indefinite number of threads. However, that number might be higher than the available number of threads in a desktop processor, so it has to be measured in practice.

Finally, we would like to remark that this paper is part of the open science effort by the authors. It is hosted in GitHub, and the paper repository hosts the data and scripts used to process them, which are in fact embedded in this paper source code using Knitr [24].

ACKNOWLEDGMENTS

This paper has been supported in part by projects TIN2014-56494-C4-3-P s (Spanish Ministry of Economy and Competitiveness) and DeepBio (TIN2017-85727-C4-2-P).

REFERENCES

[1] Enrique Alba and José M. Troya. 2001. Analyzing Synchronous and Asynchronous Parallel Distributed Genetic Algorithms. *Future Generation Computer Systems - Special issue on bioimpaired solutions to parallel processing problems* 17 (2001).

[2] J. Albert-Cruz, L. Acevedo-Martínez, J.J. Merelo, P.A. Castillo, and M.G. Arenas. 2013. Adaptando algoritmos evolutivos paralelos al lenguaje funcional Erlang. *MAEB 2013 - IX Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados* (2013).

[3] J. Albert-Cruz, J.J. Merelo, L. Acevedo-Martínez, and P. De Las Cuevas. 2014. Implementing parallel genetic algorithm using concurrent-functional languages. *ECTA 2014 - Proceedings of the International Conference on Evolutionary Computation Theory and Applications* (2014), 169–175. http://www.scopus.com/inward/record.url?eid=2-s2.0-84908690620&partnerID=40&md5=805fc82410e4f0437f27aad508e7f5fb cited By (since 1996)0.

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

[5] A. Bienz, K. Fokle, Z. Keller, E. Zulkoski, and S. Thede. 2011. A generalized parallel genetic algorithm in Erlang. In *Proceedings of Midstates Conference on Undergraduate Research in Computer Science and Mathematics.*

[6] 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

[7] J. Albert Cruz, Juan Julián Merelo Guervós, Antonio Mora García, and Paloma de las Cuevas. 2013. Adapting evolutionary algorithms to the concurrent functional language Erlang. In *GECCO (Companion)*, Christian Blum and Enrique Alba (Eds.). ACM, 1723–1724.

[8] David E. Goldberg. 1989. *Genetic Algorithms in search, optimization and machine learning.* Addison Wesley.

[9] 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

[10] Herb Sutter and James R. Larus. 2005. Software and the concurrency revolution. *ACM Queue* 3, 7 (2005), 54–62. https://doi.org/10.1145/1095408.1095421

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

[12] 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

[13] Kittisak Kerdprasop and Nittaya Kerdprasop. 2013. Concurrent Data Mining and Genetic Computing Implemented with Erlang Language. *International Journal of Software Engineering and Its Applications* 7, 3 (May 2013).

[14] Moritz Lenz. [n. d.]. *Perl 6 Fundamentals.* Springer.

[15] Fernando G. Lobo and Cláudio F. Lima. 2005. A Review of Adaptive Population Sizing Schemes in Genetic Algorithms. In *Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation (GECCO '05).* ACM, New York, NY, USA, 228–234. https://doi.org/10.1145/1102256.1102310

[16] Gabriel Luque and Enrique Alba. 2011. *Parallel Genetic Algorithms, Theory and Real World Applications.* Springer-Verlag Berlin Heidelberg, Chapter Parallel Models for Genetic Algorithms, 15–30.

[17] Juan-Julián Merelo-Guerv
'os, M. G. Arenas, J. Carpio, P. Castillo, V. M. Rivas, G. Romero, and M. Schoenauer. 2000. Evolving objects. In *Proc. JCIS 2000 (Joint Conference on Information Sciences)*, P. P. Wang (Ed.), Vol. I. 1083–1086. ISBN: 0-9643456-9-2.

[18] Juan-Julián Merelo-Guerv
'os, Gustavo Romero, Maribel García-Arenas, Pedro A. Castillo, Antonio-Miguel Mora, and Juan-Luís Jiménez-Laredo. 2011. Implementation Matters: Programming Best Practices for Evolutionary Algorithms. In *IWANN (2) (Lecture Notes in Computer Science)*, Joan Cabestany, Ignacio Rojas, and Gonzalo Joya Caparr
'os (Eds.), Vol. 6692. Springer, 333–340.

[19] L.M. Santos. 2002. Evolutionary Computation in Ada95, A Genetic Algorithm approach. *Ada User Journal* 23, 4 (2002).

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

[21] Audrey Tang. 2007. Perl 6: Reconciling the Irreconcilable. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 1–1. https://doi.org/10.1145/1190216.1190218

[22] Audrey Tang. 2007. Perl 6: Reconciling the Irreconcilable. *SIGPLAN Not.* 42, 1 (Jan. 2007), 1–1. https://doi.org/10.1145/1190215.1190218

[23] 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

[24] Yihui Xie. 2013. knitr: A general-purpose package for dynamic report generation in R. *R package version* 1, 7 (2013), 1.