Improving the algorithmic efficiency and performance of channel-based evolutionary algorithms

Juan-Julián Merelo Guervós
Universidad de Granada/CITIC
Granada,Spain
jjmerelo@gmail.com

Juan Luis Jiménez Laredo
Ri2C-LITIS, Université Le Havre
Le Havre, France
juanlu.jimenez@univ-lehavre.fr

Pedro A. Castillo
Universidad de Granada/CITIC
Granada,Spain
pacv@ugr.es

Mario García Valdez
Tecnológico Nacional de México
Tijuana, México
mario@tectijuana.edu.mx

Sergio Rojas-Galeano
Universidad Distrital Francisco José de Caldas
Bogotá, Colombia
srojas@udistrital.edu.co

ABSTRACT

Concurrent evolutionary algorithms use threads that communicate via messages. Parametrizing the work in every thread and the way they communicate results is a major challenge in its design. In this paper we work with concurrent evolutionary algorithms implemented in Perl 6, and explore different options of single-thread evolution parametrization, communication and mixing of results, showing that scalability is achieved in a multi-core environment.

CCS CONCEPTS

• **Theory of computation → Concurrent algorithms**; • **Computing methodologies → Genetic algorithms**;

KEYWORDS

Concurrent evolutionary algorithms, performance evaluation

ACM Reference Format:
Juan-Julián Merelo Guervós, Juan Luis Jiménez Laredo, Pedro A. Castillo, Mario García Valdez, and Sergio Rojas-Galeano. 2019. Improving the algorithmic efficiency and performance of channel-based evolutionary algorithms. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3319619.3322042

1 INTRODUCTION

Nowadays, concurrent programming is the best option to leverage the number of processes and threads that a multi-core processor architecture can host. These capabilities must be matched at an abstract level by concurrent languages that incorporate programming constructs intended to manage creation, execution and termination of processes, as well as new models of communication between such processes. 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 policies depending on how they deal with state, that is, data structures that could be accessed from several processes. They can be divided roughly between channel-based concurrency, with no shared nor stored state, and actor based concurrency, which stores state (in actors) but does not share it. This last model is the one used by the Perl 6 language, which is the one we are going to be using in this work.

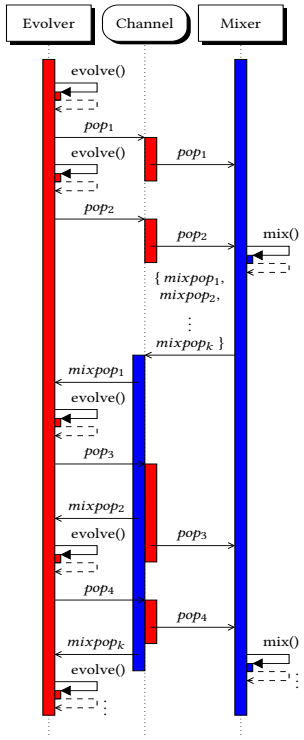

Figure 1: Concurrent EAs timeline

Previously we designed an evolutionary algorithm based on using a stateless architecture [2, 3], with different processes reacting to a channel input without changing state, and writing to the channel, but its design leaves many options open, and they have to be explored heuristically. First, we will find out what are the best parameters from the point of view of the algorithm; then, we will test several communication strategies: a lossless one that compresses the population, and a lossy one that sends a representation of population gene-wise statistics.

2 EXPERIMENTAL SETUP AND RESULTS

Building upon the design we used in previous experiments[3], here our goal was to create a system that was not functionally equivalent to a sequential evolutionary algorithms, and that followed the principle of communicating sequential processes. As in the previous

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

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3322042

papers, [2], we will use two groups of threads, one for performing the evolutionary algorithm and other for mixing populations, and two channels, one for carrying non-evolved, or generated, populations, and another for pairs of evolved populations.

The proposed system is illustrated 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 obtained and sent back to the evolutionary channel. The aim of using two channels is to avoid deadlocks; the fact that one population is always written back to the mixer channel avoids starvation in that channel.

The baseline communication model was time-costly so in this paper we introduce two different messaging strategies: *EDA*, in which the message consist of a probability distribution of each gene in the population, and *compress*, that simply bit-packs the population without the fitness into a message, using 1 bit per individual.

Similarly to our previous experiments, first we will compare these new strategies with respect to a baseline, evaluating the gap between receiving a message and activating the thread until sending the message to deactivate it. Besides, we heuristically studied other messaging strategy called *no writeback* (nw), where the mixer thread sends the individuals to the evolver channel, to undergo an additional round of evolution.

The experiments used 64-bit OneMax, a classical benchmark, it can be easily programmed in Perl 6, and allowed us to focus in the design of the relevant mechanisms of the concurrent evolutionary model; it was also used in the baseline experiments. We used the open source `Algorithm::Evolutionary::Simple` Perl 6 module.

Two evolver threads are needed to avoid starvation of the mixer thread. The generation gap was checked for the shown values, although in some cases we extended it to 4 and 64 generations. The population was sized as in previous papers using the bisection method, and the number of initial populations created and sent to the evolver channel was also designed to avoid starvation.

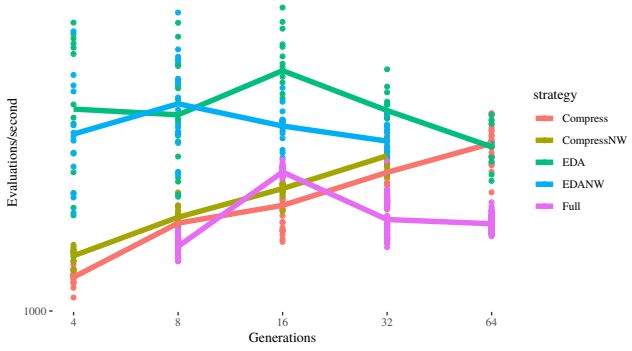

Figure 2: Evaluations per second vs. generation gap. Higher is better. Axes are logarithmic.

First, the generation gap and strategy that obtains the best number of evaluations and evaluations per second has been found, and this is shown in Figure 2 In general, the number of evaluations increases with the generation gap. More evolution without interchange with other populations implies more exploitation, and then the possibility of stagnation. However, the highest number of evaluations per second are achieved by the EDA strategies, as shown in the figure. This EDA-style communication strategy finds the best balance between number of evaluations and communication speed mainly due to the compactness of its messages, but also due to the fact that it is non-elitist and might avoid stagnation or dominance of the population by a super-individual.

However, the intention of concurrent evolutionary algorithms is to leverage the power of all threads and processors in a computer, so unlike in previous papers, we must find a version of the algorithm that speeds up with the number of threads. After many tests, eventually the scaling strategy was simply to divide the total population by the number of threads. This resulted in a decrease of wallclock time from 2 to 4 threads, and an additional increase of evaluations/second up to 8 threads. However, since smaller population brings about decreased diversity, more evaluations were needed and time actually increased from 4 threads up.

3 CONCLUSIONS

In this paper we explored the parameter space in a concurrent evolutionary algorithm looking for the combination that yields the best speedup performance, without affecting its algorithmic effectiveness. In order to do so, the size of messages interchanged within the channel has been redesigned using the distribution of probabilities as a representation of the population. Different messaging strategies has been also tested.

Experiments show that the number of generations that the population undergoes must be kept to a small number. The results also indicate that the EDA strategy is the fastest, with a relatively low impact in the number of evaluations, as the messages are the most compact. Finally, obtained results have shown that simultaneous threads running an evolutionary algorithm via population splitting do increase the number of simultaneous evaluations, leading the new concurrent evolutionary algorithm to improve the performance in comparison to the equivalent single thread evolutionary algorithm. However, the total time does not decrease in the same proportion due to the effect of the division of population.

ACKNOWLEDGMENTS

This paper has been supported in part by projects DeepBio (TIN2017-85727-C4-2-P) and TecNM Project 5654.19-P.

REFERENCES

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

[2] Juan J. Merelo and José-Mario García-Valdez. 2018. Going Stateless in Concurrent Evolutionary Algorithms. In *Applied Computer Sciences in Engineering*, Juan Carlos Figueroa-García, Eduyn Ramiro López-Santana, and José Ignacio Rodriguez-Molano (Eds.). Springer International Publishing, Cham, 17–29.

[3] Juan J. Merelo and José-Mario García-Valdez. 2018. Mapping Evolutionary Algorithms to a Reactive, Stateless Architecture: Using a Modern Concurrent Language. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '18).* ACM, New York, NY, USA, 1870–1877. https://doi.org/10.1145/3205651.3208317