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

Creating a concurrent and stateless version of an evolutionary algorithm implies changes in its behavior. From the performance point of view, the main challenge is to balance computation with communication, but from the algorithmic point of view we have to keep diversity high so that the algorithm is not stuck in local minima. In a concurrent setting, we will have to find the right balance so that improvements in both fields do not cancel out. This is what we will be doing in this paper, where we explore the space of parameters of a population based concurrent evolutionary algorithm to find out the best combination for a particular problem.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

ACM proceedings, LaTeX, text tagging

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, **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Concurrent programming adds a layer of abstraction over the parallel facilities processors and operating systems to offer a high-level interface that allows the user to program code that might be executed in parallel either in threads or in different processes [**?**].

Different languages offer different facilities for concurrency at the primitive level, and mainly differ on how they deal with shared state, that is, variables that are accessed from several processes. Actor-based concurrency [**?**] eliminates shared state by introducing a series of *actors* that store state and can change it; on the other hand, channel based concurrency follows the *communicating sequential processes* methodology [**?**], which is effectively stateless, with different processes reacting to channel input without changing state, and writing to these channels.

This kind of concurrency is the one implemented by many modern languages such as Go or Perl 6 [**?**]. However, the statelessness of the implementation asks for a change in the implementation of any algorithm. In particular, evolutionary algorithms need to change to fit an architecture that creates and processes streams of data using functions that do not change state.

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

For instance, the EvAg model [**?**] is a locally concurrent and globally parallel evolutionary algorithm that leaves the management of the different agents (i.e. threads) to the underlying platform scheduler and displays an interesting feature: the model is able to scale seamlessly and take full advantage of CPU threads. In a first attempt to measure the scalability of the approach experiments were conducted in [**?**] for a single and a dual-core processor showing that, for cost functions passing some milliseconds of computing time, the model was able to achieve near linear speed-ups . This study was later on extended in [**?**] by scaling up the experimentation to up to 188 parallel machines. The reported speed-up was ×960 which is beyond the linear ×188 that could be expected if local concurrency were not taken into account.

The aforementioned algorithm used a protocol that worked asynchronously, leveraging its peer-to-peer capabilities; in

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

general the design of concurrent EAs has to take into account the communication/synchronization between processes, which nowadays will be mainly threads. Although the paper above was original in its approach, other authors targeted explicitly multi-core architectures, such as Tagawa [?] which used shared memory and a clever mechanism to avoid deadlock. Other authors [?] actually use a message-based architecture based in the concurrent functional language Erlang, which separates GA populations as different processes, although all communication takes place with a common central thread.

In our previous papers [? ?], we presented the proof of concept and initial results with this kind of stateless evolutionary algorithms, implemented in the Perl 6 language. These evolutionary algorithms use a single channel where whole populations are sent. The (stateless) functions read a single population from the channel, run an evolutionary algorithm for a fixed number of generations, which we call the *generation gap* or simply *gap*, and send the population in the final generation back to the channel. Several populations are created initially, and a concurrent *mixer* is run which takes populations in couples, mixes them leaving only a single population with the best individuals selected from the two merged populations. This *gap* is then conceptually, if not functionally, similar to the *time to migration* in parallel evolutionary algorithms (with which concurrent evolutionary algorithms have a big resemblance).

We did some initial exploration of the parameter space in [?]. In these initial explorations we realized that the parameters we used had an influence at the algorithmic level, but also at the implementation level, changing the wallclock performance of the algorithm.

In this paper we will explore the parameter space systematically looking particularly at two parameters that have a very important influence on performance: population size and generation gap. Our intention is to create a rule of thumb for setting them in this kind of algorithms, so that they are able to achieve the best performance.

We will present the experimental setup next.

2 EXPERIMENTAL SETUP AND RESULTS

In principle, we will work with two threads, one for mixing the populations and another for evolution. This setup, while being concurrent, allows us to focus more on the basic features of the algorithm, which is what we are exploring in this paper.

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 [?], 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 [?], 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 **??**.

3 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, transmiting 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.

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