# Serverless Evolutionary Computation

Bailey Eccles
School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast, UK
beccles01@qub.ac.uk

***Abstract*—This preliminary research article presents the concept of mapping evolutionary algorithms to the cloud-based serverless architecture in the aim to evolve these highly parallelisable algorithms beyond the hardware limitations of traditional systems. Firstly, an introduction to the problem discusses the motive and proposed methodology for migrating these algorithms to the serverless paradigm. Secondly, a systematic literature review explores articles related to the research area.**

***Index Terms*—Serverless computing, high performance computing, distributed computing, stateless algorithms, evolutionary algorithms.**

## I. Introduction

EVOLUTIONARY algorithms (EA) are a subset of generic population-based metaheuristic optimisation algorithms that model biological evolution mechanisms; reproduction, mutation, recombination, and selection. Typically these mechanisms are applied to a random population of individuals then evaluated by a fitness function. Fittest individuals are selected for the next iteration of recombination and mutation that breed offspring to replace the least-fit individuals. Continuous iterations of breeding then selecting improve the population's average fitness. After a time limit, number of iterations, or until a sufficient fitness is achieved; an individual is selected as a candidate solution to the problem.

The process described above of is a brief overview for one type of EA, Genetic algorithms (GA). A simple GA comprises of two genetic operators: **recombination**, which is the crossover of genetic information from parent individuals to generate child individuals, and **mutation**, which replace genetic information with randomised values. For a population of $n$ individuals: $n$ mutations or $n/2$ recombinations are required to evolve the population by one generation. The expected execution time for one generation depends on the population size as each genetic operator takes a relatively small amount of time to generate an individual. As a result, GA implementations can be made highly concurrent by distributing the population across multiple processors.

On the contrary, depending on the complexity of the fitness function, population evaluation is the most computationally expensive process often resulting in higher latencies than other parts of a GA. Repeated evaluation of the population can lead to significant bottlenecks during run time as the subsequent generation must wait until the current generation is evaluated. This limitation is considered a major drawback for high performance GAs.

As such, increasing the throughput of a simple GA can be achieved as follows:

1) Maximise the concurrency of genetic operators
2) Minimise the cost of population evaluation

Many approaches to Parallel Genetic Algorithms (PGAs) already exist [1] such as Island GAs [2], Pool-based GAs [3] and GPU-based GAs [4]. However, these approaches are generally problem-dependant or architecture-dependant. One of the main benefits of GAs is their customisability and the ability to solve complex problems with simple implementations. This article proposes the idea of exploiting the auto-scaling capabilities of serverless architecture to create GAs that are flexible, highly configurable, and scale seamlessly to any problem domain.

Although actually involving servers, the serverless paradigm abstracts software implementations from hardware constraints. Since 2016, well-established cloud providers such as Amazon Web Services (AWS), Google Cloud and Microsoft Azure offer Functions-as-a-Service (FaaS) platforms that allow abstract code execution without provisioning a server. For the purposes of creating a high performance serverless GA, the AWS serverless platform AWS Lambda offer the greatest scalability out of all providers (1,000+ concurrent executions per function, Up to 1536 MB of memory per function [1]).

Using AWS Lambda, the realisation of a serverless GA will involve mapping the genetic operators and population evaluation as serverless functions. Functions may communicate to each other using common methodologies such as REST over HTTP, JSON or simple messages, however, it is important to note that serverless functions are stateless (no information is retained by the function after execution). Statelessness is a non-issue for genetic operators as parent individuals may be sent to the function, processed, then a child individual sent back using the discussed event-based methods. However, non-serverless population controllers for evaluating fitness and selection of individuals require a state for management of the population. Migrating these to serverless functions is nontrivial. No recent research has proposed a functional stateless population controller. A potential approach to add state is integrating a high performance in-memory caching system such as AWS DAX that fully supports AWS Lambda.

The aim of a serverless GA is to create an auto-scaling, high performance optimisation system that can outperform and scale beyond equivalent traditional non-serverless implementations.

[1] https://dzone.com/articles/comparing-serverless-architecture-providers-aws-az

## II. Literature Review

José-Mario et al. [5] proposes an event-based distributed EA framework, KafkEO. Introduced as a proof of concept, KafkEO has no working implementation or prototype. Intended to be functionally equivalent to an island GA, one merit is that KafkEO supports the option of swapping the search step with any population-based optimisation algorithm such as particle swarm optimisation (PSO). Likewise, this preliminary research article proposed an approach that supports swapping the search step with any EA that can be translated to a serverless function. However, a major drawback in the design of KafkEO is that search step receives triggers and sends responses to a monolithic population controller service outside the serverless framework. Acknowledged in the article, the stateful population controller would be susceptible to creating bottlenecks in the response messaging queue. The reason for the stateful population controller is to allow migration between sub-populations (a property of island-based GAs). A stateless population controller could be introduced with some stateful properties to facilitate a fully serverless model so that the population controller could also leverage the scaling capabilities of serverless architecture which minimises the effect of bottlenecks between the search step and population evaluation. As a proof of concept, no experimental results are provided for KafkEO, however, a critical observation of the design would guess the scaling capabilities of serverless have a negligible effect on the parallel performance of the GA due to the sequential time spent evaluating individuals in the stateful population controller.

Juan J. et al. [6] investigates different models of concurrency in stateless architecture. Implemented in Perl6, a language build for concurrency and functional programming. Perl6 is similar to other languages such as Go, Scala and Erlang. The implementation has two variants. Firstly, a Individual-level concurrency model distributes function operations across the population's set of individuals which is an atypical model for PGAs, however, it is the same concurrency model as this preliminary research article's proposed approach. Secondly, a Population-level concurrency model, similar to a canonical island-based GA, where parallelisation is achieved via concurrent executions of sub-populations. Both variants of the Perl6 implementation were compared against a sequential baseline evolutionary algorithm to solve the 64bit onemax problem [2]. The results conclude that the individual-level concurrent variant can solve the 64bit onemax problem in one order of magnitude less evaluations than both the population-level concurrent and sequential baseline. The article does not compare the run time of each algorithm to reach the candidate solution to the problem, therefore, it was not disclosed if the Perl6 implementations generated the candidate solution quicker than the sequential baseline. The experiments were also carried out with a relatively small population size of 256; fine-grain PGAs can achieve a greater parallel speedup on larger populations than sequential implementations. The approach proposed in this preliminary research article will be able to take full advantage of over 1'000 concurrent function executions per generation allowing for a shorter run time to solve the same problem or solve the same problem for a larger bit size in the same amount of time as the desktop counterpart.

Wlodzimierz et al. [7] experiments with the scalability and upper-limit of a distributed fitness evaluation service. The article proposes the best way to scale distributed fitness evaluation is with an in-memory caching system. Results conclude that data sets of up to $\sim$1 GB can achieve a relatively large parallel speedup at scale over sequential systems. However, these experiments were carried out on relatively old AWS EC2 architecture [3] and used an outdated monolithic software framework Apache Spark [4]. This preliminary research article proposes distributed fitness evaluation on modern serverless architecture where it is expected these results will perform the same or better when scaling a serverless GA.

## Appendix A
## Research Progress

AWS Lambda natively supports a variety of programming languages for building serverless functions and applications. It is important to review the current language options as each offer different strengths and weaknesses depending on their particular language paradigm and system type checking on a virtual environment. Java has a large overhead when provisioning a virtual environment to run as a serverless function due to the added computation layers of the Java Virtual Machine (JVM) and static type checking. Interpreted, dynamic type checking languages without any virtualisation bloat such as Python, do not have as much overhead when running as a serverless function. Language overhead primarily affects the performance of a serverless function during a cold-start; where AWS Lambda assigns the function a virtual environment to run on. For a short time interval after provisioning, subsequent requests to the same serverless function will be assigned the same virtual environment, therefore, the provisioning of virtual elements and compilation of static languages has already occurred, thus minimising the execution time. These are known as a warm-start serverless function calls.

Fig. 1 shows the relative performance of a cold-start for various languages compared to the same function running on physical hardware. Serverless Java is over 1600x slower on a cold-start, however, a warm-start decreases this to $\sim$ 60x slower. Dynamic languages average 35% better on a cold-start, however, are many times slower than the baseline. All languages greatly benefit from a warm-start. A key take-away from this experiment is that serverless functions should be kept alive for as long as possible to leverage the increased performance of a warm-start. However, a warm-start is not on par with the baseline, therefore, further research should be carried out to increases the performance further. Fig. 2 shows the maximum memory usage for each language for the same function. No implementation utilises anywhere near the AWS Lambda function maximum of 1536 MB, however, this should be continuously evaluation throughout further code development in case large functions reach the limitation.

[2] https://tracer.lcc.uma.es/problems/onemax/onemax.html

[3] Deprecated `m1.medium`, `m1.large`, and `m1.xlarge` EC2 instances

[4] https://spark.apache.org/

## APPENDIX B
## REFERENCE GRAPHS

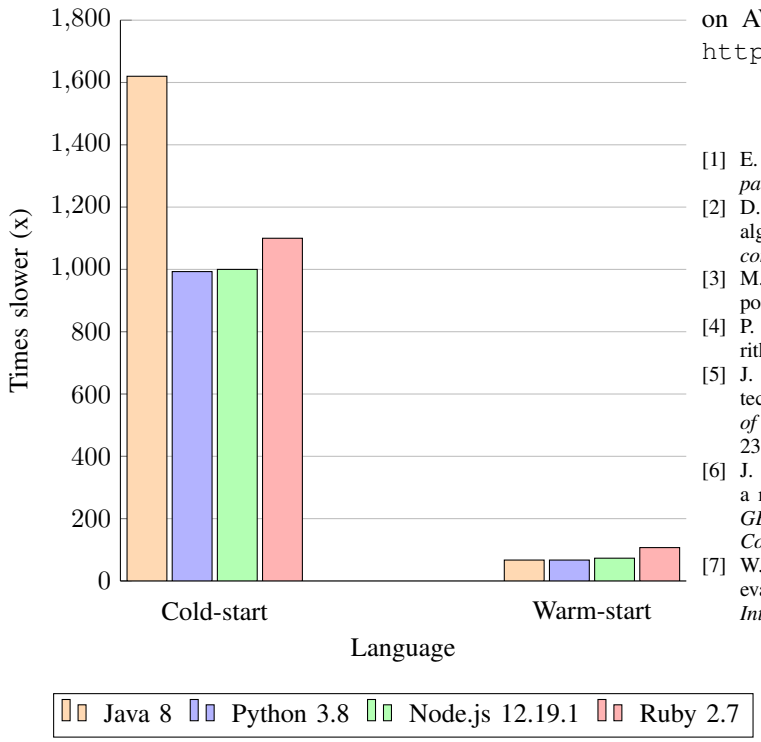


Fig. 1. Box plot that represents the relative execution time (times slower) between a reference java implementation and various cold-start and warm-start serverless function language implementations

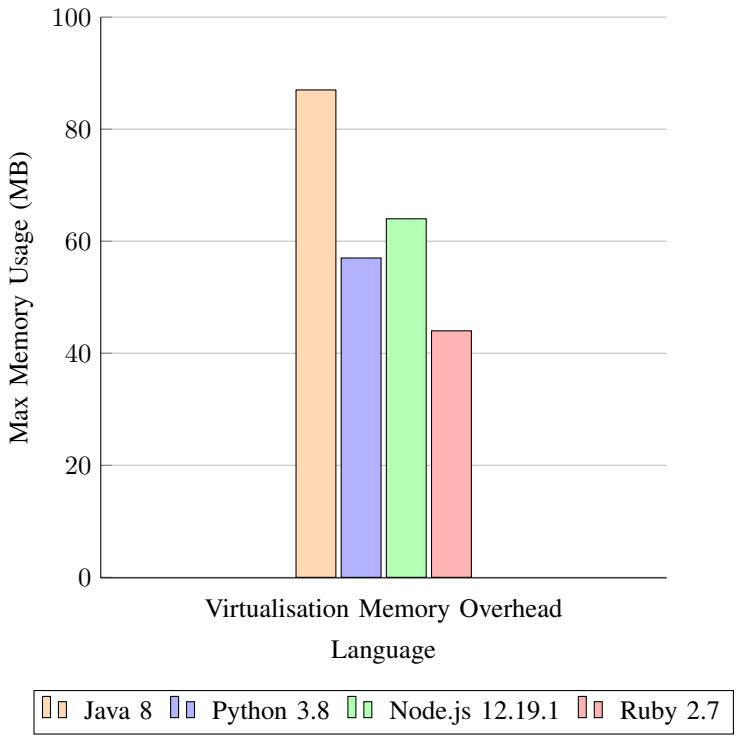


Fig. 2. Box plot that represents serverless virtualisation memory overhead for various serverless function language implementations

## APPENDIX C
## CODE PROGRESS

All code implementations and experiments were carried out on AWS Lambda. Current code progress is available here: `https://github.com/123Bailey123/SEC`.

## REFERENCES

[1] E. Cantú-Paz, "A survey of parallel genetic algorithms," *Calculateurs paralleles, reseaux et systems repartis*, 1998.

[2] D. Whitley, S. Rana, and R. Heckendorn, "The island model genetic algorithm: On separability, population size and convergence," *Journal of computing and information technology*, 1999.

[3] M. García-Valdez, L. Trujillo, and J. Merelo, "The evospace model for pool-based evolutionary algorithms," *Journal of Grid Computing*, 2015.

[4] P. Pospichal and J. Jaros, "Gpu-based acceleration of the genetic algorithm," *GECCO competition*, 2009.

[5] J. García-Valdez and J. Merelo-Guervós, "A modern, event-based architecture for distributed evolutionary algorithms," *GECCO '18: Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 233–234, 2018.

[6] J. Merelo and J. García-Valdez, "Mapping evolutionary algorithms to a reactive, stateless architecture: using a modern concurrent language," *GECCO '18: Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 1870–1877, 2018.

[7] W. Funika and P. Koperek, "Towards a scalable distributed fitness evaluation service," *Parallel Processing and Applied Mathematics: 11th International Conference*, pp. 493–502, 2015.