Introducing an Event-Based Architecture for Concurrent and Distributed Evolutionary Algorithms

Juan J. Merelo Guervós[1(✉)] and J. Mario García-Valdez[2]

[1] Universidad de Granada, Granada, Spain
jmerelo@geneura.ugr.es
[2] Instituto Tecnológico de Tijuana, Tijuana, BC, Mexico
mario@tectijuana.edu.mx

Abstract. Cloud-native applications add a layer of abstraction to the underlying distributed computing system, defining a high-level, self-scaling and self-managed architecture of different microservices linked by a messaging bus. Creating new algorithms that tap these architectural patterns and at the same time employ distributed resources efficiently is a challenge we will be taking up in this paper. We introduce KafkEO, a cloud-native evolutionary algorithms framework that is prepared to work with different implementations of evolutionary algorithms and other population-based metaheuristics by using micro-populations and stateless services as the main building blocks; KafkEO is an attempt to map the traditional evolutionary algorithm to this new cloud-native format. As far as we know, this is the first architecture of this kind that has been published and tested, and is free software and vendor-independent, based on OpenWhisk and Kafka. This paper presents a proof of concept, examines its cost, and tests the impact on the algorithm of the design around cloud-native and asynchronous system by comparing it on the well known BBOB benchmarks with other pool-based architectures, with which it has a remarkable functional resemblance. KafkEO results are quite competitive with similar architectures.

Keywords: Cloud computing · 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

1 Introduction

Cloud computing is increasingly becoming the dominant way of running the server side of most enterprise applications nowadays, the same as the browser is the standard platform for the client side. Besides the convenience of the pay-as-you-go model, it also offers a way of describing the infrastructure as part of

A. Auger et al. (Eds.): PPSN 2018, LNCS 11101, pp. 399–410, 2018.
https://doi.org/10.1007/978-3-319-99253-2_32

the code, so that it is much easier to reproduce results and has been a boon for scientific computing. However, programming the cloud means that monolithic applications, that is, applications built on a single stack of services that communicate by layers, are no longer an efficient architectural design for scientific workflows. Cloud architectures favor asynchronous communication over heterogeneous resources, and shifting from mostly sequential and monolithic to an asynchronously parallel architecture will also imply important reformulation of the algorithms in order to take full advantage of these technologies. Cloud-native applications add a layer of abstraction to the underlying distributed computing system, seamlessly integrating different elements in a single data flow, allowing the user to just focus on code and service connections. Services are *native* points in this new architecture, departing from a monolithic or even distributed paradigm to become a loosely collection of services, in fact *microservices* [19], which in many cases are stateless, reacting to some event and *living* only while they are doing some kind of processing. Reactive systems not only allow massive scaling and independent deployment they are also more economical than other monolithic options. Platform as a service (PaaS) or even Container as a Service (CaaS) approaches need to be running all the time in order to maintain their state, so they are paid for their size and time they remain active. At any rate, while one of the main selling points of Functions as a Service (FaaS) is their ultra-fast activation time, from our point of view their most interesting feature is the fact that they provide stateless processing. An important caveat of stateless processing is that algorithms must be adapted to this fact and turned, at least in part, into a series of stateless steps working on a data stream. It is also taken to an atomic extreme with the so-called serverless architectures [20], which allow vendors and users to deploy code as single, stateless functions, that get activated via *rules*, *triggers* or explicitly, reacting to events consumed from a message queue. The first commercial implementation of this kind of architecture was released by Amazon with its Lambda product, to be closely followed by releases by Azure (Microsoft) and Google and OpenWhisk, an open source implementation released by IBM [2].

In this paper we want to introduce KafkEO, a serverless framework for evolutionary algorithms and other population-based systems. The main design objective is to leverage the scaling capabilities of a serverless framework, as well as create a system that can be deployed on different platforms by using free software. Our intention has also been to create an algorithm that is functionally equivalent to an asynchronous, parallel, island-based, EA, which can use parallelism and at the same time reproduce mechanisms that are akin to migration. The island-based paradigm is relatively standard in distributed EA applications, but in our case, we have been using it since it allows for better parallelism and thus performance, at the same time it makes keeping diversity easier while needing fewer parameters to tune.

We will examine the results of this framework using the first five functions of the Noiseless Black-Box-Optimization-Benchmarking (BBOB) testbed [10] part of the COCO (COmparing Continuous Optimisers) platform for comparisons

of real-parameter global optimisers [10]. The framework is compared against another cloud-ready parallel pool based implementation. The implementation is also free software and can be downloaded from GitHub. The rest of the paper is organized as follows. Next we present the state of the art in cloud implementation of evolutionary algorithms, to be followed in Sect. 3 by an introduction to the serverless architecture we will be using as well as our mapping of the evolutionary algorithm to it. Section 4 will present the result of performing experiments with this proof of concept; finally in Sect. 5 will discuss the results, present conclusions and future lines of work.

2 State of the Art

In general, scientific computing has followed the trends of the computing industry, with new implementations published almost as soon as new technologies became commercially available, or even before. There were very early implementations of evolutionary algorithms on transputers [21], the world wide web [5] and the first generation of cloud services [12,16,17]. However, every new computing platform has its own view of computing, and in many cases that has made evolutionary algorithms move in new directions in order to make them work better in that platform while keeping the spirit of bio-inspiration. For instance, most evolutionary algorithms work in a synchronous way; although there were very early efforts to create asynchronous EAs [6], in general generations proceed one after the other and migration happens in all islands at the same time. However, this mode of working does not fit well with architectures that are heterogeneous and dynamic, which is why there have been many efforts from early on to adapt EAs to this kind of substrate [1,3,22].

This kind of internet-native applications later on transitioned to using Service-Oriented Architectures (SoA) [14]. While monolithic, that is, including all services in a single computing node and application, SoA were better adapted to heterogeneous environments by distributing services across a network using standard protocols. Several authors implemented evolutionary algorithms over them [8,13,15]. However, scaling problems and the extension of cloud deployment and services had made this kind of architectures decline in popularity.

In general, frameworks based in SoA also tried to achieve functional equivalence with parallel or sequential versions of EAs. There is the same tension between functional equivalence and new design in new, cloud based approaches to evolutionary algorithms. Salza and collaborators [16,17] explicitly and looking to optimize interoperability claim that there is very little need to change "traditional" code to port it to the cloud, implicitly claiming this *functional equivalence* with sequential evolutionary algorithms.

Besides these implementations using well known cloud services, there are new computation models for evolutionary algorithms that are not functionally equivalent to a canonical EA, but have proved to work well in these new environments. Pool based EAs, [4], with a persistent population that can be tapped to retrieve single individuals or pools of them and return evaluated or evolved

sub-populations, have been used for new frameworks such as EvoSpace [9], and proved to be able to accommodate all kinds of ephemeral and heterogeneous resources.

In the serverless, event-based architectures we are going to be targeting in this paper, there has been so far no work that we know of. Similar setups including microservices have been employed by Salza et al. [17]; however, the proposed serverless system adds a layer of abstraction to event-based queuing systems such as the one employed by Salza by reducing it to functions, messages and rules or triggers. We will explain in detail these architectures in the next section.

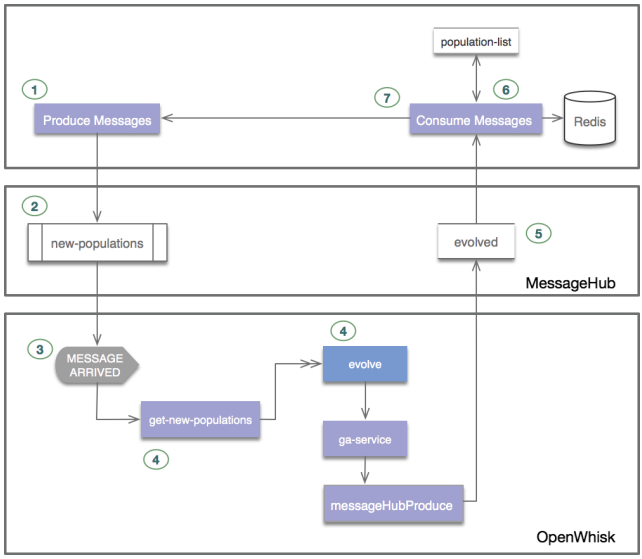

Fig. 1. Chart showing the general picture of the layers of a serverless architecture, including the messages and services that constitute KafkEO, with labels indicating message routes and the software components used for every part.

3 Event-Based Architectures and Implementing Evolutionary Algorithms over Them

Microservice architectures share the common trait of consisting of several services with a single concern, that is, providing a single processing value, in many cases stateless, and coupled using lightweight protocols such as REST and messaging buses that carry information from every service to the next. In this case, we are going to be using IBM's BlueMix service, which includes OpenWhisk as a serverless framework and MessageHub, a vendor implementation of the Kafka messaging service; this last one gives name to the framework we are presenting, called KafkEO (EO stands for Evolving Objects).

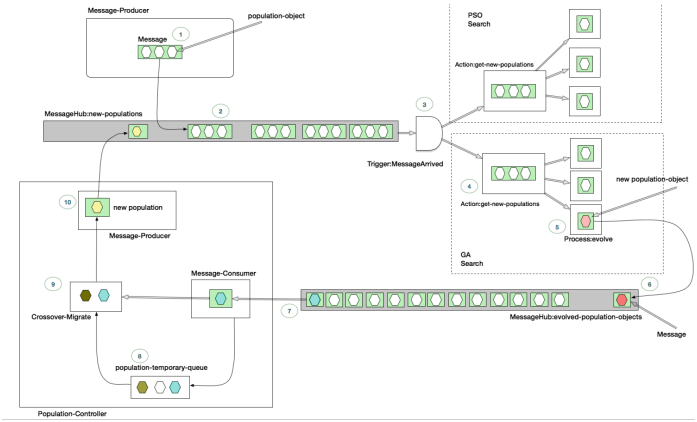

Fig. 2. A flow diagram of KafkEO, showing message routes, MessageHub topics and the functions that are being used.

The main reason for choosing OpenWhisk and Kafka is availability of resources, but also the fact that all parts of the implementation are open source and can be deployed in desktop machines or other cloud providers by changing the configuration. It is also a good practice to implement free software using free software, making it widely available to the scientific community.

The layers and message flow in the application are shown in Fig. 1, which also includes the evolutionary components. We will focus for the time being in the general picture: a serverless architecture using a messaging service as a backbone, which in this case takes the shape of the Kafka/MessageHub service. These messages are produced and consumed by a service, which can also store them in an external database for their later use; in general, messaging systems are configured to keep messages only for a certain amount of time, and they disappear after that. Messaging queues are organized in *topics* and every topic uses a series of *partitions*, which can be increased for bigger throughput; the functions, hosted in OpenWhisk, execute *actions* triggered by the arrival of new messages; these actions also produce new messages that go back to the MessageHub to continue with the message loop. If all this is hosted in a cloud provider, as it is the case, the MessageHub service will be charged according to a particular cost structure, with partitions taking the most part of the cost, while messages have a relatively small impact.

The evolutionary algorithm mapped over this architecture is represented in Fig. 2. The main design challenge is to try and map an evolutionary algorithm to a serverless, and then stateless, architecture. That part is done in points 1 through 5 of Fig. 2. The beginning of the evolution is triggered from outside the serverless framework (1) by creating a series of Population objects, which we pack (2) to a message in the new-populations *topic*. Population objects are the equivalent to islands or samples in EvoSpace. If Population object is a self-contained population of individuals, represented as a JSON structure.

The arrival of a new population package sets off the MessageArrived trigger (3), that is bound to the actions that effectively perform a small number of generations. In this case we give as an example a GA and a PSO algorithms, although only the GA has been implemented for this paper. Any number of GA algorithms (*actions*) can be triggered in parallel by the same message, and new actions can be triggered while others are still working; this phase is then self-scaling and parallel by design.

Population objects are extracted from the message and, for each, a call to an *evolve* process is executed in parallel. The *evolve* process consists of two sequential *actions* (5), first, the *GA Service* function that runs a GA for a certain number of generations, producing a new evolved object, which is then sent to the second action called *Message Produce* responsible of sending the object to the evolved-population-objects message queue. The new Population object (6) includes the evolved population and also metadata such as a flag indicating whether the solution has been found, the best individual, and information about each generation. With this metadata a posterior analysis of the experiment can be achieved or simply generating the files used by the BBOB Post-processing scripts.

This queue is polled by a service outside the serverless framework, called Population-Controller. This service needs to be stateful, since it implements a buffer that needs to wait until several populations are ready to then mix them (in step #9 in Fig. 2) to produce a new population, that is the result of selection and crossover between several populations coming from the evolved-population-objects message queue. Eventually, these mixed populations are returned to the initial queue to return to the *serverless* part of the application. Another task of the Population-Controller is to start and stop the experiment. The service must keep the number of Population objects received, then after a certain number is reached, the controller stops sending new messages to the new-populations *topic*. It is important to note, that because of the asynchronous nature of the system, several messages could still arrived after the current experiment is over. The controller must only accept messages belonging to the current process.

This *merging* step before starting evolution takes the place of the *migration* phase and allows this type of framework to work in parallel, since several instances of the function might be working at the exact same time; the results of these instances are then received back by every one of the instances.

In fact, this kind of system would be more functionally equivalent to a pool-based architecture [4], since the queue acts as a pool from where populations are taken and where evolved populations return. Actually, the pool becomes a *stream* in this case, but in fact the pool also evolves, changing its composition, and has a finite size just like the pool. Since pool-based architectures have already proved they work with a good performance, we might expect this type of architecture, being functionally equivalent, to be at least just as efficient and the latter, and better adapted to a cloud-native application.

In this phase where we are creating a proof of concept, there is a single instance of this part. For the time being, it has not been detected as a bottleneck,

although eventually, when the number of functions are working in parallel, it might become one. There are several options for overcoming this problem, the easiest of which is to add more instances of this Population-controller. These instances will act in parallel, processing the message queue at different offsets and contributing to population diversity. This will eventually have its influence in the results of the algorithm, so it is better left as future work.

Since we are running just a few functions, the amount of code of KafkEO is quite small compared with other implementations. We use DEAP for all the evolutionary functions, which are written in Python and released in GitHub under the GPL license.

4 Experiments and Results

In this section we compare the performance of KafkEO against an implementation of the EvoSpace [9] pool-based architecture, using the first five functions of the Noiseless BBOB testbed [10], which are real-parameter, single-objective, separable functions, namely: Sphere, ellipsoidal, which is highly multimodal, Rastrigin, Buche-Rastrigin, and the purely lineal function called linear slope. It is expected that the two algorithms achieve similar results as they are functionally equivalent. The EvoSpace implementation follows the basic EvoSpace model in which EvoWorkers asynchronously interact with the population pool by taking samples of the population to perform a standard evolutionary search on the samples, to then return newly evolved solutions back to the pool.

EvoWorkers were implemented in Python with the same code as KafkEO and using DEAP [7] for the GA service function. The code is in the following GitHub repository: https://hidden.com. Before each experiment, a script initializes the population on the server, creating the number of individuals specified by the *Pool Size* parameter, this size depends on the dimension of the problem according to the BBOB testbed. When starting each EvoWorker, the following parameters are used: first, the *Sample Size* indicating the number of individuals the worker would take from the server on each interaction, then the *Iterations per Sample* parameter specifies the number of generations or iterations the worker algorithm will run before sending back to the server the resulting population. Finally, the number of times an EvoWorker will take, evolve and return a sample, is indicated by the *Samples per Worker* parameter. The number of EvoWorkers instantiated for the experiment is given by the *GA Workers* parameter. The EvoSpace parameters are shown in Table 1. These parameters are set for each dimension and they indicate the effort in number of evaluations. In both experiments the maximum number of evaluations is $10^5 \cdot D$. For instance, for $D = 2$, the maximum number of evaluations is $200,000$ which is obtained by multiplying the parameters int the first column of Table 1: $50 \cdot 100 \cdot 20 \cdot 2$. Also both algorithms limit the search space to $[-5, 5]^D$.

On the other hand, the parameters used for KafkEO are shown in Table 2. Every function runs an evolutionary algorithm for the shown number of iterations and with the population size also shown. The number of initial messages act as

Table 1. EvoWorker setup parameters,

Dimension	2	3	5	10	20	40
Iterations per Sample	50	50	50	50	50	50
Sample Size	100	100	100	200	200	200
Samples per Worker	20	30	25	25	25	25
GA Workers	2	2	4	5	8	16
Pool Size	250	250	500	1000	2000	4000

an initial trigger, being thus equivalent to the number of parallel functions or workers; this is the tunable parameter used for increasing performance when the problem dimension, and thus difficulty, increases; the population size is also increased, so that initial diversity is higher. Please note that every population is generated randomly, so that the population size would have to be multiplied by the number of initial messages to get to the initial population involved in the experiment. The effort is limited by the maximum number of messages consumed by the *Population-Controller* from the evolved-population-objects message queue. The maximum number is calculated by multiplying the *Maximum Iterations* and *Initial Messages* parameters. Again for a $10^5 \cdot D$ maximum number of evaluations.

Table 2. KafkEO parameters for the BBOB benchmark. Dimensions are the independent variable, the rest of the parameters are changed to adapt to the increasing difficulty.

Dimension	2	3	5	10	20	40
Iterations	50	50	50	50	50	50
Population Size	100	100	100	200	200	200
Initial Messages	2	2	4	5	8	16
Maximum Iterations	2	2	4	5	8	16

The evolutionary algorithm implemented in KafkEO used the same code, also delegating the evolutionary operations to the standard DEAP library, written in Python [7], using 12 for tournament size, a Gaussian mutation with sigma = 0.05 and a probability between 0.1 and 0.6, plus two point crossover with probability between 0.8 and 1; these are the default parameters. In particular, the tournament size injects a high selective pressure which is known to decrease diversity. The system also allows to set different parameters for every instance; in this proof of concept only two parameters were randomly set, *Mutation Probability* uniformly random in the [.1, .6] range, and *Crossover Probability* random on [.8, 1]. This is one deviation from the standard evolutionary algorithm, but has been proved in the past to provide good results without needing to fine tune different parameters [18].

The experiments were performed during the month of January using a paid IBM BlueMix subscription. The totality of experiments costed about $12. Most of the cost is due to the MessageHub *partitions*, that is, the hosted messaging service itself. The amount paid for the messages in the BlueMix platform is less than one dollar in total; messages are paid by the hundreds of thousands delivered, and are actually not the most expensive part of the implementation of the algorithm. Partitions are essential for a high throughput; a messaging queue will be able to process as many messages as the partitions are able to get through in parallel; this means that cost will scale with the number of messages in a complex way, not simply linearly, and design decisions will have to be taken. The baseline is that the best option is to maximize the number of messages that can be borne by a particular partition, and try to minimize the number of partitions to avoid scaling costs.

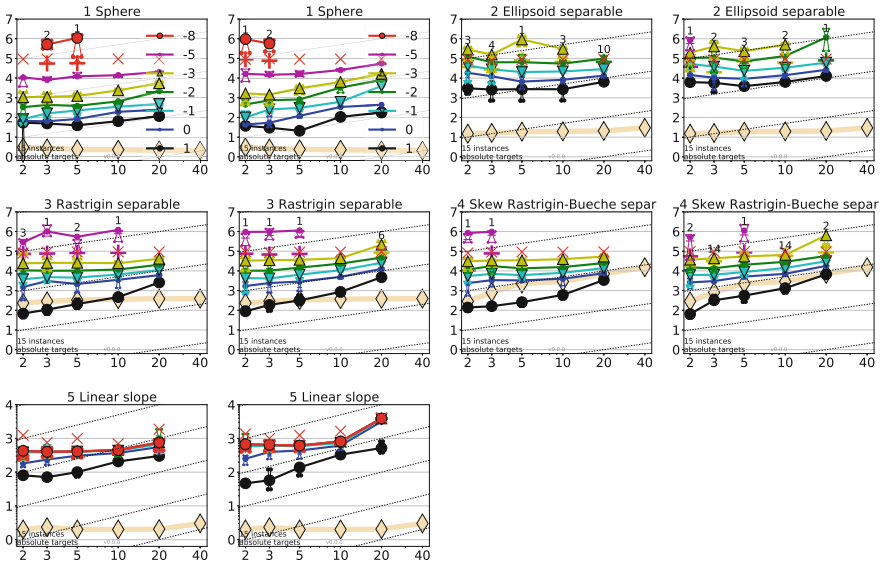

Fig. 3. Scaling of the running time with dimension to reach certain target values Δf. Lines: average runtime (aRT); Cross (+): median runtime of successful runs to reach the most difficult target that was reached at least once (but not always); Cross (×): maximum number of f-evaluations in any trial. Notched boxes: interquartile range with median of simulated runs. All values are divided by dimension and plotted as $\log_{10}$ values versus dimension. Shown is the aRT for fixed values of $\Delta f = 10^k$ with k given in the legend. Numbers above aRT-symbols (if appearing) indicate the number of trials reaching the respective target. The light thick line with diamonds indicates the best algorithm from BBOB 2009 for the most difficult target. Horizontal lines mean linear scaling, slanted grid lines depict quadratic scaling. Odd columns (1, 3): EvoSpace; even columns (2, 4): KafkEO.

The results of the comparison are shown in Fig. 3, which follow the classical BBOB 2009 format, which includes the amount of effort devoted to finding a certain fitness level and time needed to do it. Figure 3 was generated by the post-processing script from COCO [10] used in the Black-box Optimization Benchmarking workshop series. The EvoSpace and KafkEO results are shown side by side, only for the sake of comparison, since we are only interested for the time being in the baseline performance of the proof of concept.

The results obtained show that the basic Genetic Algorithm implemented in KafkEO does not perform very well against the testbed, specially when compared against other nature inspired algorithms like PSO or other hybrid approaches [11]. However, both implementations, shown side by side, reach similar results with the same effort; and the results in this case have been obtained with fewer parameters, with out the need to specify an initial pool size and without tuning the evolutionary algorithm parameters.

This is a problem that pool-based algorithms have: we need to specify the initial number of individuals to place in the pool and have the burden of always keeping a minimum number of individuals in the pool. This is not the case in KafkEO, because there is no need to have a repository for the population. However, population size and the number of generations turned in by every instantiation of the functions have to be tuned, which is something that will have to be left as future work.

5 Conclusions

This paper is intended to introduce a simple proof of concept of a serverless implementation of an evolutionary algorithm. The main problem with this algorithm, shared by many others, is to turn something that has state (in the form of loop variables or anything else) into a stateless system. In this initial proof of concept we have opted to create a stateful *mixer* outside the serverless (and thus stateless) platform to be able to perform *migration* and mixing among populations. A straightforward first step would be to parallelize this service so that it can respond faster to incoming evolved populations; however, this scaling up should be done by hand and a second step will be to make the architecture totally serverless by using functions that perform this mixing in a stateless way. This might have the secondary effect of simplifying the messaging services to a single topic, and making deployment much easier by avoiding the desktop or server back-end we are using now for that purpose.

The proof of concept is a good adaptation of an evolutionary algorithm to the serverless architecture, with a performance that is comparable, in terms of number of evaluations, to pool-based architectures. Even if results right now are not competitive, the scalability of the architecture and also the possibilities it offers in terms of tuning parameters for the algorithm, even using heterogeneous functions tapping the same *topic* (channel), offer the chance of improving running time as well as the algorithm itself in terms of number of evaluations. This is an avenue that we will explore in the near future. The whole set of experiments, done

in the cloud with a desktop component, took more than running a single desktop experiment using EvoSpace. However, scaling was lineal with problem difficulty, which at least mean that we are not adding an additional level of complexity to the algorithm and might indicate that horizontal or vertical scaling would solve the problem. This kind of scaling also indicates that the stateful part, run in a desktop, has not for this problem size become a bottleneck. Even so, we consider that it is essential to create an algorithm architecture that will be fully serverless and, thus, stateless.

Other changes will go in the direction of testing the performance of the system and computing the cost, so that we can increase the former without increasing the latter. Since there is room for increasing parallelism, we will try different ways of obtaining better algorithmic results by making a parameter sensitivity analysis, including population size, length of evolution runs, and other algorithmic parameters. Once those algorithmic baselines have been set, we will experiment with different metaheuristics such as particle swarm optimization, or even try for heterogeneous functions with different evolutionary algorithm parameters, with the purpose of reducing the number of parameters to set at the start.

Acknowledgments. Supported by projects TIN2014-56494-C4-3-P (Spanish Ministry of Economy and Competitiveness) and DeepBio (TIN2017-85727-C4-2-P).

References

1. Atienza, J., Castillo, P.A., García, M., González, J., Merelo, J.: Jenetic: a distributed, fine-grained, asynchronous evolutionary algorithm using Jini. In: Wang, P.P. (ed.) Proceedings of JCIS 2000 (Joint Conference on Information Sciences), vol. I, pp. 1087–1089 (2000). ISBN: 0-9643456-9-2
2. Baldini, I., et al.: Cloud-native, event-based programming for mobile applications. In: Proceedings - International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016, pp. 287–288 (2016)
3. Baugh, J.W., Kumar, S.V.: Asynchronous genetic algorithms for heterogeneous networks using coarse-grained dataflow. In: Cantú-Paz, E., et al. (eds.) GECCO 2003. LNCS, vol. 2723, pp. 730–741. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45105-6_88
4. Bollini, A., Piastra, M.: Distributed and persistent evolutionary algorithms: a design pattern. In: Poli, R., Nordin, P., Langdon, W.B., Fogarty, T.C. (eds.) EuroGP 1999. LNCS, vol. 1598, pp. 173–183. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48885-5_14
5. Chong, F.S., Langdon, W.B.: Java based distributed genetic programming on the internet. In: Banzhaf, W., et al. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, p. 1229. Morgan Kaufmann, Orlando, 13–17 July 1999. Full text in technical report CSRP-99-7
6. Coleman, V.: The DEME mode: an asynchronous genetic algorithm. Technical report, University of Massachussets at Amherst, Department of Computer Science (1989). uM-CS-1989-035
7. Fortin, F.A., Rainville, F.M.D., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: evolutionary algorithms made easy. J. Mach. Learn. Res. **13**, 2171–2175 (2012)

8. García-Sánchez, P., González, J., Castillo, P.A., Arenas, M.G., Merelo-Guervós, J.: Service oriented evolutionary algorithms. Soft Comput. **17**(6), 1059–1075 (2013)
9. García-Valdez, M., Trujillo, L., Merelo, J.J., Fernández de Vega, F., Olague, G.: The EvoSpace model for pool-based evolutionary algorithms. J. Grid Comput. **13**(3), 329–349 (2015). https://doi.org/10.1007/s10723-014-9319-2
10. Hansen, N., Auger, A., Mersmann, O., Tusar, T., Brockhoff, D.: COCO: a platform for comparing continuous optimizers in a black-box setting (2016). arXiv preprint arXiv:1603.08785
11. Hansen, N., Auger, A., Ros, R., Finck, S., Pošík, P.: Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009. In: Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 1689–1696. ACM (2010)
12. Merelo-Guervós, J.J., Arenas, M.G., Mora, A.M., Castillo, P.A., Romero, G., Laredo, J.L.J.: Cloud-based evolutionary algorithms: an algorithmic study. CoRR abs/1105.6205, 1–7 (2011)
13. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: The design, usage, and performance of GridUFO: a grid based unified framework for optimization. Future Gener. Comput. Syst. **26**(4), 633–644 (2010)
14. Papazoglou, M.P., van den Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. VLDB J. **16**(3), 389–415 (2007). https://doi.org/10.1007/s00778-007-0044-3
15. Rodríguez, L.G., Diosa, H.A., Rojas-Galeano, S.: Towards a component-based software architecture for genetic algorithms. In: 2014 9th Computing Colombian Conference (9CCC), pp. 1–6, September 2014
16. Salza, P.: Parallel genetic algorithms in the cloud. Ph.D. thesis, University of Salerno, Italy (2017). https://goo.gl/sDx6mY
17. Salza, P., Hemberg, E., Ferrucci, F., O'Reilly, U.M.: cCube: a cloud microservices architecture for evolutionary machine learning classification. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 137–138. ACM (2017)
18. Tanabe, R., Fukunaga, A.: Evaluation of a randomized parameter setting strategy for island-model evolutionary algorithms. In: 2013 IEEE Congress on Evolutionary Computation (CEC), pp. 1263–1270. IEEE (2013)
19. Thönes, J.: Microservices. IEEE Softw. **32**(1), 116–116 (2015)
20. Varghese, B., Buyya, R.: Next generation cloud computing: new trends and research directions. Future Gener. Comput. Syst. **79**, 849–861 (2018). Cited by 2
21. Voigt, H.-M., Born, J., Santibañez-Koref, I.: Modelling and simulation of distributed evolutionary search processes for function optimization. In: Schwefel, H.-P., Männer, R. (eds.) PPSN 1990. LNCS, vol. 496, pp. 373–380. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0029778
22. Zorman, B., Kapfhammer, G.M., Roos, R.S.: Creation and analysis of a JavaSpace-based distributed genetic algorithm. In: PDPTA, pp. 1107–1112 (2002)