Future Generation Computer Systems 116 (2021) 234–252

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A container-based cloud-native architecture for the reproducible execution of multi-population optimization algorithms

Mario García Valdez [a,*], Juan J. Merelo Guervós [b]

[a] Department of Graduate Studies, Instituto Tecnológico de Tijuana, Tijuana BC, Mexico
[b] Department of Computer Architecture and Technology, Universidad de Granada, Granada, Spain

ARTICLE INFO

Article history:
Received 30 January 2020
Received in revised form 19 October 2020
Accepted 31 October 2020
Available online 6 November 2020

Keywords:
Multi-population
Nature-inspired algorithm
Parallel genetic algorithms
Cloud-computing
Event-driven architecture

ABSTRACT

Splitting a population into multiple instances is a technique used extensively in recent years to help improve the performance of nature-inspired optimization algorithms. Work on those populations can be done in parallel, and they can interact asynchronously, a fact that can be leveraged to create scalable implementations based on, among other methods, distributed, multi-threaded, parallel, and cloud-native computing. However, the design of these cloud-native, distributed, multi-population algorithms is not a trivial task. Using as a foundation monolithic (single-instance) solutions, adaptations at several levels, from the algorithmic to the functional, must be made to leverage the scalability, elasticity, (limited) fault-tolerance, reproducibility, and cost-effectiveness of cloud systems while, at the same time, conserving the intended functionality. Instead of an evolutive approach, in this paper, we propose a cloud-native optimization framework created from scratch, that can include multiple (population-based) algorithms without increasing the number of parameters that need tuning. This solution goes beyond the current state of the art, since it can support different algorithms at the same time, work asynchronously, and also be readily deployable to any cloud platform. We evaluate this solution's performance and scalability, together with the effect other design parameters had on it, particularly the number and the size of populations with respect to problem size. The implemented platform is an excellent alternative for running locally or in the cloud, thus proving that cloud-native bioinspired algorithms perform better in their "natural" environment than other algorithms, and set a new baseline for scaling and performance of this kind of algorithms in the cloud.

1. Introduction

In the last decades, nature-inspired optimization algorithms have been successfully applied to solve many complex real-world problems [1]. These algorithms can be broadly grouped into evolutionary algorithms (EAs) [2] and swarm intelligence (SI) [3], as well as other categories; popular EAs are Genetic Algorithms (GAs) [4,5], Genetic Programming (GP) [2], Gray Wolf Optimization (GWO) [6] and Differential Evolution (DE) [7], while examples of (SI) [3] are particle swarm optimization (PSO) [8] and ant colony algorithms (ACO) [9].

Besides being, all of them, population-based, they also share the common characteristic of creating an initial set of random candidate solutions which is later used by a nature-inspired metaheuristic to generate a new set of candidates; there are parts of this process that can be easily done in parallel. For instance, the fitness of each individual can be (in general) evaluated independently of others, and similarly, each population could evolve in isolation. Researchers use the term multi-population based methods when referring to techniques using many populations as part of their optimization strategy. Since earlier works, researchers have been proposing some form of parallelization [10] to increase these algorithms' scalability by adding some form of interaction between populations running in isolation. The island model was one of the first techniques proposed for parallelization, which led to early proofs of speed-up [11,12]. Since then, other population-based algorithms have adopted the idea. Researchers have found additional benefits besides the execution speed; these include avoiding a premature convergence and maintaining the diversity of the global population [13].

Moreover, parallel implementations can run in two different ways: synchronous and asynchronous. When operations run in parallel, but every node must maintain synchrony with the other operations, they all need to finish before moving to the next step. For instance, in a master/worker model, masters need to wait for all workers to complete their operations before moving to the next iteration. In contrast, in an asynchronous execution, operations are not synchronized with each other, so there

* Corresponding author.
E-mail addresses: mario@tectijuana.edu.mx (M. García Valdez), jmerelo@geneura.ugr.es (J.J. Merelo Guervós).

https://doi.org/10.1016/j.future.2020.10.039
0167-739X/

is no time lost waiting for other populations to arrive at the synchronization point; results coming from other workers are incorporated as soon as they are received. Following the previous example, now the master operation can continue to the next iteration, even if a worker has not yet finished its work. We can find many asynchronous algorithms [14,15] reporting benefits in execution time and scalability. In the particular case of asynchronous and cloud-native multi-population algorithms, recent works propose an asynchronous communication through a central repository [16,17] or message queues [18,19]. In this work, we follow this practice.

The trend concerning parallel execution multi-population algorithms goes from earlier hardware-based implementations using transputers [11] through multi-core systems [20,21] to multi-threaded [22] ones. Moreover, recently, the focus has been on exploiting a higher number of processing units by using GPUs [23,24], or distributed systems, including web-based [17] systems, map-reduce [25] implementations, grids [26,27], voluntary computing systems [28,29], and, more importantly, cloud computing [18,30–32].

Cloud computing has become the standard way of running enterprise applications. Not only because of the convenience of the pay-as-you-go model or the non-existent sunk cost of physical facilities, but also because it offers a way of describing the infrastructure as part of the code, so that it is much easier to reproduce results and this has been a boon for scientific computing. However, cloud computing has also been evolving, going from simply putting old-style monolithic applications on virtual machines executing on an external data center to microservices [33] and serverless architectures [34,35] that favor the parallel and asynchronous communication of heterogeneous resources. In the process, a new methodology for designing *cloud-native applications* has been created, with brand new methods and techniques [36]. In these architectures, services are seen as independent processing nodes, departing from monolithic or distributed paradigms to becoming a loosely coupled collection of *stateless functions* [37], that react to events. Systems developed with the patterns outlined above have the properties of a reactive system [38], and they are generally more flexible, fault-tolerant, and scalable.

Researchers and algorithm designers are willing to adapt to the cloud; however, when developing their multi-population algorithms, they face additional challenges:

- They need to change their current solutions to a reactive architecture in which processes communicate with each other by exchanging messages asynchronously and react to a continuous stream of data. One of the options is to consider populations as messages to be modified asynchronously by functions.
- They need to establish a workflow of local development and prototyping while having the option of deployment to cloud services, thus using modern methodologies and technologies.
- They must log experimental results and be able to replicate the experiments.
- They must consider additional parameters that can affect the monetary cost or the performance of the execution — for instance, the format and size of messages, the number of worker processes and their capabilities.

Beyond the choice of design methodology and implementation, researchers need to consider additional issues [39] when designing efficient multi-population algorithms. These include the number and size of populations, the interaction between them, the search area of each population, and the search strategy and parametrization for each population. We must take into account these high-level issues when designing a parallel architecture.

Considering these factors, in this work, we present EvoSwarm, a cloud-native, container-based application that reactively processes isolated and heterogeneous populations. EvoSwarm is released with a free license and available in a public GitHub repository.[1]

This paper extends our earlier publication on the topic [19], and we highlight the main contributions as follows: We have proved that cloud-native applications can easily accommodate multi-paradigm bioinspired algorithms with little efforts of parametrization and that a combination of algorithms has better speed-up and performance than any single one taken independently.

We have proved this in the following steps:

- First, we present the design and implementation of a reactive container-based application for the asynchronous execution of multi-population algorithms. The source code and example container definitions are publicly available.
- Second, we propose a new method for the deployment and execution of multiple experiments by specifying the infrastructure as part of an experiment definition in both local or cloud environments, facilitating the reproduction of experimental results.
- Third, the application is compatible with the COCO benchmark framework, allowing researchers to compare their algorithms' performance against other works.
- Fourth, we present an empirical study to validate our application's applicability, measuring the execution time and speed-up.
- Lastly, we present a comparison between homogeneous and an ensemble of multi-populations, using Genetic Algorithms (GAs) and Particle Swarm Optimization (PSO) using a benchmark for continuous function optimization.

We organized the paper as follows: First, in Section 2, we present a background of the fundamental issues of integrating nature-inspired optimization and multi-population methods. Section 3 presents state of the art relevant to our work. In Section 4, we present the proposed method and the container-based application in Section 5 and the workflow for reproducible experiment design in Section 6. Section 7 describes the empirical evaluation design to assess the method's effectiveness, concerning scalability and the heterogeneous populations' performance. Finally, we offer the conclusions of this paper and suggestions for future work in Section 8.

2. Multi-population methods

Multi-population based methods divide the original population into smaller populations or islands, with every population carrying out the algorithm independently, with synchronous or asynchronous communication with the rest of the islands [39]. This relative isolation maintains an overall diversity since each population will search in a particular area, at least between communications. The recombination mentioned above (mixing) or migration between populations is needed to avoid a premature convergence of candidate solutions since smaller populations are known to perform better for a given problem than bigger populations [40,41]. However, it gives them the added advantage of parallel operation. Additionally, and in some cases, multi-population algorithms scale better than expected due to the interaction between the algorithm and the parallelism of the operation [42].

[1] https://github.com/mariosky/EvoSwarm.

However, in most cases, algorithms applied to each population are homogeneous; but as long as this parallel operation is event-based, other population-based algorithms could be easily integrated. That is why several works based on multi-population are heterogeneous, integrating various optimization algorithms, and often performing better than single-population or homogeneous optimization algorithms [41,43].

Heterogeneous algorithms add another degree of freedom to the parameter tuning problem; because some parameters affect the accuracy of the solution and the convergence speed of the algorithms as they tip the balance between exploration and exploitation of the search space. On the other hand, current studies show that by having a high number of populations interacting in parallel, the effect of each population's parameters is compensated by those selected in other populations [40,44].

Combining multiple algorithms with different parameters, interacting with each other at the same time, can benefit from the strengths of each. For instance, a genetic algorithm could find a promising global solution that is not optimal while another algorithm, more suitable for a local search, finds the global optimum. This approach has been followed extensively in recent years with success [13,45,46]. Moreover, there is a need for frameworks, architecture, and implementation models that can allow researchers the development of new parallel, asynchronous, heterogeneous, and parameter-free algorithms in a scalable way.

3. State of the art

Although containers look like merely a method for easy shipping of applications, at first sight, they are much more than that. One of the first features that make them interesting is their ephemerality: fast startup and teardown time make them usable for machine-independent and cloud deployment of simple, ephemeral, and maybe stateless functions. The full realization of Docker-based architectures has led to the creation of a series of patterns that together define the cloud-native application development [47].

However, this has been a slow realization in the evolutionary algorithms field, from the origins where Salza and Ferrucci [48,49] proposed an architecture that carried evolutionary algorithms to the cloud, and which were developed further in their next paper [50]. These papers started to introduce cloud-native aspects such as the use of messaging queues and CoreOS as an operating system designed from the ground up for container utilization in the area of evolutionary algorithms. However, the way these containers are managed, is rather classical from the distributed EC point of view: a master–worker approach, communicated using RabbitMQ (a messaging queue), where replicated workers perform the tasks in parallel. They called their approach AMQPGA, inspired by the protocol, AMQP, used by RabbitMQ. Other authors have also followed this classical approach by using other "classical" EA topologies in the cloud: Dziurzanski et al. [51] adapts an island model to a container architecture, in a system that is functionally equivalent to classical models, with the advantage of the auto-scaling of islands, and being easy to deploy and run.

Later approaches used microservices [52] and provided the automation of the whole workflow [18], achieving speed-up by creating an abstraction layer over the evolutionary algorithm services. This research bumped into some problems, since speed-up is limited, mainly due to the non-automatic scaling of services and the presence of a single master, which is the one that runs the evolutionary algorithm. However, this master–worker architecture can be turned around with other kinds of evolutionary architectures, such as pool-based systems [16,31,53] that were a better match to the inherently heterogeneous nature of cloud-based architectures. Pool-based systems are closer to serverless systems [37], as they pull populations from the pool and run whole algorithms, instead of just doing the evaluation. The pool is a shared data storage for results and can be accessed in an asynchronous way, which suits better the nature of cloud-native systems.

4. Proposed method

In this section, we present the design and a container-based implementation of the model we propose to execute multi-population-based algorithms. We follow the requirements outlined in Section 2 and the general design principles of cloud-native applications we have mentioned earlier; we intend to steer clear of a direct translation of "classical" GA architectures to the cloud and design from the ground up a cloud-native method. As a first step, we describe the general architecture and then every component in detail.

The most basic data structure used throughout the system is the population, and at a higher level, we can view the system as a continuous stream of populations flowing from one process to the other. By "continuous stream", we mean that ideally, everything must happen asynchronously without components needing to wait for others. In this kind of system, this streaming functionality is normally implemented by using a message queue system. For instance, if one process needs to communicate a population to another, it must pack the population into a message and then send the message to the queuing system without waiting for a response; this means that after the process pushes the message, it continues its execution without waiting for a result. On the other end, the recipient subscribes to the message queue by defining an event handler method that will be triggered when a message is pushed to it. At each end of the queue, the two components are Producers that push messages to the queue and Consumers that pull them. This pattern is an essential component of a reactive architecture and makes it highly scalable; one reason for this is the use of functions with no secondary effects, that is, stateless functions. These functions do not need to read, keep, or modify data outside of the method. If Consumers are implemented using stateless functions, there is no difference between having one or many copies of the same function simultaneously pulling work from the queue. There are no side effects, including unwanted resource locking problems resulting from this concurrency. Based on these general principles, we next define the proposed model.

4.1. Event-driven model

Based on a reactive architecture we proposed in our previous work [19], we now describe the general architecture shown in Fig. 1. Again, we can explain the model using the analogy of producers and consumers of messages. First, we can see two queues, one labeled `Input` and the other one `Output`. In the diagram, push operations on a queue are represented by solid arrows connecting to the left side of the queue box, and pull or pop operations as solid arrows leaving from the right side. The architecture has at least four processes indicated in the diagram as swimlanes: First, the `Setup` process, responsible for reading a configuration file and creating the initial populations. Second, the `Controller` process, responsible for the migration between populations and keeping track of the algorithm's iterations. The `Message Queue` process runs the `Input` and `Output` queues. Finally, there is at least one `Stateless Function` process responsible for running the isolated algorithms. In the example, two processes `PSO` and `GA`, are shown.

The algorithm starts with the `Setup` process pulling a configuration message from the `Experiments` queue (not shown in the diagram). The configuration message includes all the parameters

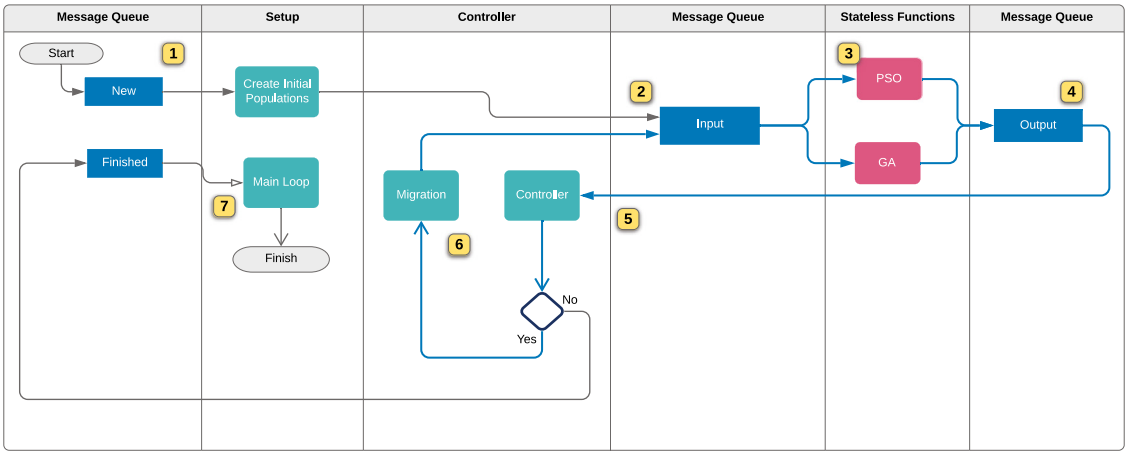


Fig. 1. The proposed general architecture, showing each process in a swimlane, message-dataflow, message queues, and high-level tasks for each process.

needed to execute the algorithm, the number of populations, the number of individuals, and the number of iterations of the algorithm. We give more details about the configuration data structure in Section 6. Once the configuration is read, we can follow the path of messages as follows:

1. In this step, the specified number of populations are created according to the parameters found in the configuration structure. The population at this moment is just static data, including each individual inside. Each population includes a metadata section where its algorithm and execution parameters are specified. For instance, for a GA, the mutation rate, type of crossover operator, and other values are indicated.
2. Each population is then pushed to the `Input` queue so that they can be consumed by stateless functions responsible for the search's execution.
3. One or more `Stateless functions` are constantly pulling population messages from the `Input` queue. They take the current state of the population and run the algorithm for a certain number of iterations.
4. Once they finish the execution, the population state is again packed with additional metadata about the algorithm's execution. The resulting populations are now pushed to the Output queue. Once finished, another population is pulled from the queue.
5. The `Controller` process is responsible for keeping track of the progress of the search. It pulls current populations from the `Output` queue, inspects the metadata. If an optimal solution has been found or the maximum number of iteration has been reached, it signals the execution to stop.
6. Otherwise, it passes the stream of messages to a migration process, where populations are mixed. New populations are generated from this migration and again pushed to the Input queue to continue in a loop. The `Controller` is also responsible for logging the metadata received with the messages.

This reactive architecture has the following advantages:

- An important aspect of the proposal is the decoupling of the population and the population-based algorithm. It is common that in a classic island-based algorithm, each island is executed in a separate processing node, i.e., in a virtual machine, CPU, or thread. In this case, we can have just one processing node or many nodes, each running with a different search strategy or parameters.
- Also, the reactive controller gives designers more control over the multi-population algorithm. In this process, designers can dynamically change the number of populations, population parameters, and migration details on-the-fly.
- Another advantage is that algorithm designers have many options for implementing this simple architecture. The same components can be implemented as a single multi-threaded program or as a highly scalable serverless cloud application. Most modern languages include constructs for asynchronous programming using queues or channels for multi-threaded execution.

On the other hand, we have several caveats: It is more costly to move entire populations as messages than passing only specific individuals from a process to another. Designers must consider this cost when working with large individuals, and if possible, send populations using several messages, use compression, or adjust the populations' size. Pool-based algorithms also suffer from this drawback.

4.2. Comparison with other cloud-based works

Next, we compare the message-driven architecture we presented against four multi-population-based algorithms found in the literature. We center the comparison on the coupling and communication between the main components: populations, processing nodes, and algorithms. In Fig. 2, we show the main components of a message-driven architecture. We can see that the algorithm and populations are separated, and the `Population` queue keeps the state of populations. While algorithms only need the populations as parameters for their execution.

In contrast, in the classic island model shown in Fig. 3, algorithms are population methods, and the same processing node keeps both the algorithm process and the state of the population. Other processing nodes follow the same configuration and execution form, and they only pass specific individuals between them. In this model, adding additional processing nodes, on-runtime, can be more difficult because they need to have an addressing mechanism or simply be aware of these new nodes. This is solved in part in Peer to Peer (P2P) evolutionary algorithms [54]. However, coupling still exists, meaning that every node must run the whole evolutionary algorithm and hold full copies of its subpopulation.

A typical design pattern used to alleviate the above drawbacks is using a central repository or pool of individuals available to all processing nodes. In Fig. 4 we show the main components of a pool-based multi-population algorithm. Although algorithms and

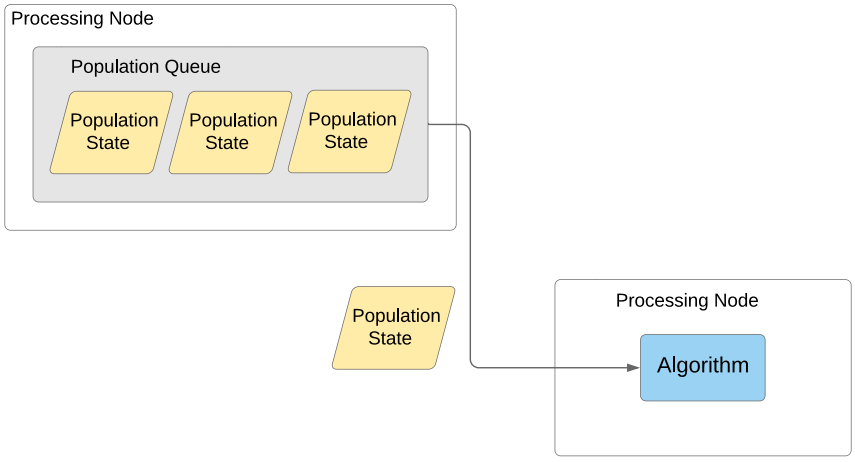


Fig. 2. Population state and dataflow between processing nodes of a message-based algorithm.

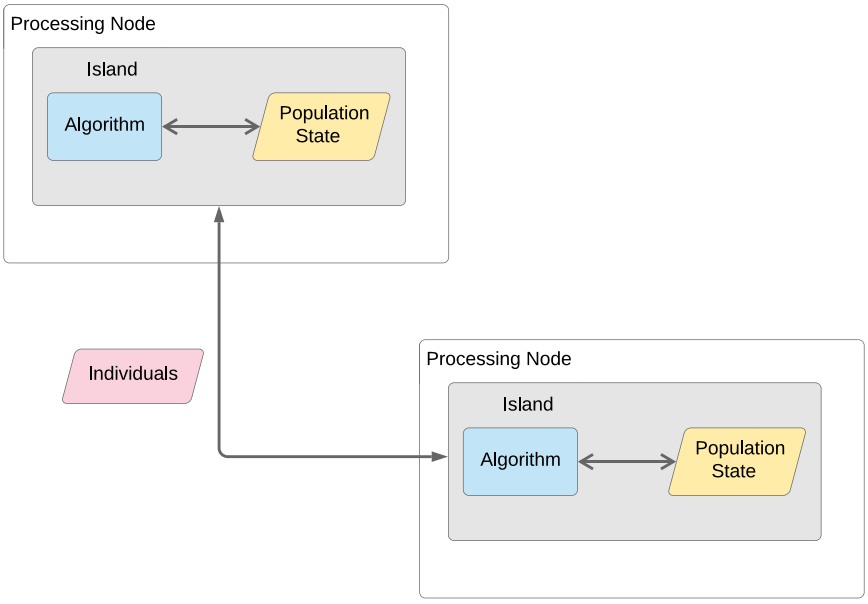


Fig. 3. Population state and dataflow between processing nodes of a classic Island-based multi-population algorithm.

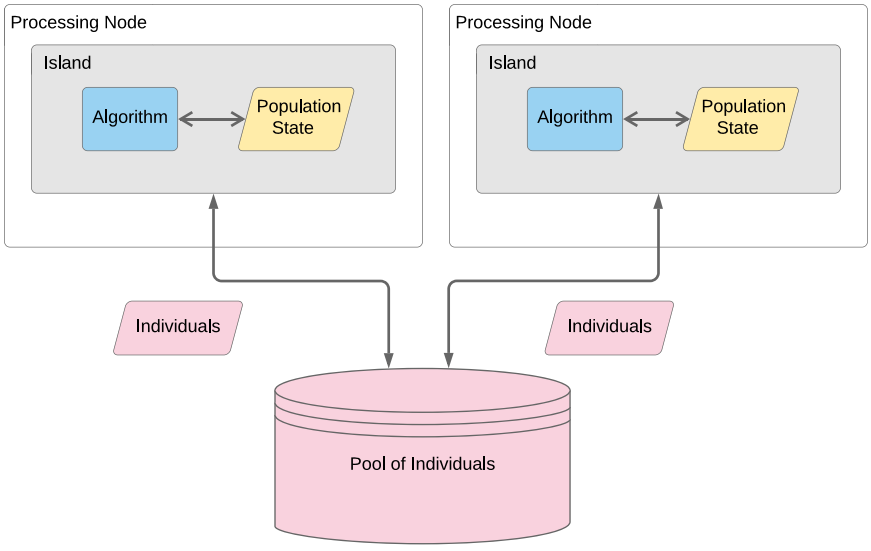


Fig. 4. Population state and flow of data between processing nodes of a Pool-based multi-population algorithm.

population state remain coupled, now, processing nodes do not communicate directly with each other. Instead, they interchange individuals with the pool. Communication between nodes is not affected if the system adds or removes nodes because they do not know about each other. Since the state of the global population is stored in external processing nodes, the system needs additional communication and processing for keeping track of each population.

We show another pool-based approach in Fig. 5. In this design, the global population is stored in a central repository, while isolated algorithms take random samples of the global population and use this temporal population as parameters. An advantage of this design is that the sampling provides a type of migration between isolated populations. A problem found is that when a processing node returns a population to the pool, the population's state is lost. Nevertheless, having the algorithm decoupled opens the possibility of implementing the system using serverless functions.

5. Experimental design of a container-based application

This section proposes a design of a reactive container-based application for executing multi-population-based optimization experiments. We followed the cloud-native design patterns highlighted in previous sections, and again we can go back to Fig. 1 and use it as a guide for explaining the main components and their interactions. The containers used in this design will be explained next.

5.1. Message queue container

In this work, we implemented all message queues in the Redis memory store. Early cloud architectures [50] used queues for communication, although in that case, RabbitMQ was the chosen tool. Redis is faster and can act as a data store from where we can control the application's state. Each queue is a Redis List object, and we use the `LPOP` (left pop) and `RPOP` (right pop) commands for a queue like behavior. In those cases where we needed a blocking behavior, i.e., when a process needs to wait until a message is available, we used the blocking versions `BLPOP` and `BRPOP`. Redis in-memory operations are very fast, and all of the above operations have a time complexity of $\mathcal{O}(1)$. We use the official `redis:alpine` container image from DockerHub.

5.2. Controller container

The controller is an essential component of the architecture (see Fig. 1) because it is responsible for maintaining the evolutionary loop. It takes newly evolved populations from the `Output Queue`, mixes them, and produces new messages from the result. At the same time, it must keep track of two conditions for ending the loop: the number of messages it has received reaches a maximum value, or the error of the best solution goes below a specific threshold. Finally, it must filter out remnant messages from other experiments. Messages from other experiments can remain in the queues because of the asynchronous nature of the system. We chose to implement the controller in Python but using an API specialized in asynchronous event-driven programming over data streams. The open-source library is called Reactive Extensions (ReactiveX) for Python[2] and it is based on the Observer pattern [55] and functional programming.

In ReactiveX, an `observer` object subscribes to an `Observable` instance. Observables emit a sequence of items, and all the subscribed observers react to each emission. The ReactiveX library includes several reactive operators that we can use to transform and combine sequences of items. These operators provide reactive extensions that allow us to compose asynchronous sequences together in a declarative manner. In Fig. 6, we use a marble diagram to represent the composition of `Observables` and reactive extensions operators. We represent the timeline of an `Observable` as a horizontal arrow, which indicates that time flows from left to right. In the diagram, items are represented as marbles. The position of each marble indicates the point in time when they were emitted by the `Observable`. Reactive operators are represented as text boxes, showing the transformation to be applied. Typically, a transformation results in another `Observable`, again emitting new results. This approach is more elegant for asynchronous programming than nested callbacks, which are more difficult to code and debug.

Now we proceed to explain the reactive implementation of the controller. In Fig. 6, we have the following composition of Observables:

1. The controller continually pulls new messages from the `Output` message queue. These messages are instantly emitted by the `consumed_messages Observable`. This particular example shows that the stream receives two populations from another experiment, and these are shown as two red marbles. The filter operator removes all messages that belong to a different experiment; in this case, we are only interested in blue marbles.
2. The `max_iterations` parameter indicates the number of populations that are going to be accepted. When this number of messages is reached, we must end the loop. In this example, `max_iterations` $= 6$, the take operator assures that only six messages are received. After the `Observable` emits the sixth message, it triggers the completed event.
3. Many observers can be subscribed to the `valid_messages Observable` because it emits all the valid items. At the moment, there are two additional methods subscribed that we are not showing, one for logging and the other for monitoring the search.
4. The `buffer_with_count(3)` operator waits until it receives three valid messages to emit a single message that contains a list with the previous three. The `population_mixer` method requires that list to mix them. In our previous work, we needed a local buffer for storing a certain number of populations to mix them with others. This design has the advantage of not needing extra memory, and it integrates better with the reactive paradigm. A possible disadvantage can be that it only mixes contiguous populations, but this can be mitigated with a larger buffer.
5. The `population_mixer` receives a list of three populations, let us say [A, B, C], and calls the migration method shown in Algorithm 1 for [A, B], [B, C] and [A, C]. The migration algorithm sorts both populations and generates a new population containing the best half from each. This migration method is similar to those used successfully in previous works on pool-based algorithms Fig. 4. Finally, the method pushes the new populations to the `messages Observable`. From there, a `publish(population)` observer is responsible for pushing the newly generated populations back to the `Input` message queue.

In the next section, we follow the algorithm flow and describe the `worker` containers' implementation details.

[2] https://github.com/ReactiveX/RxPY.

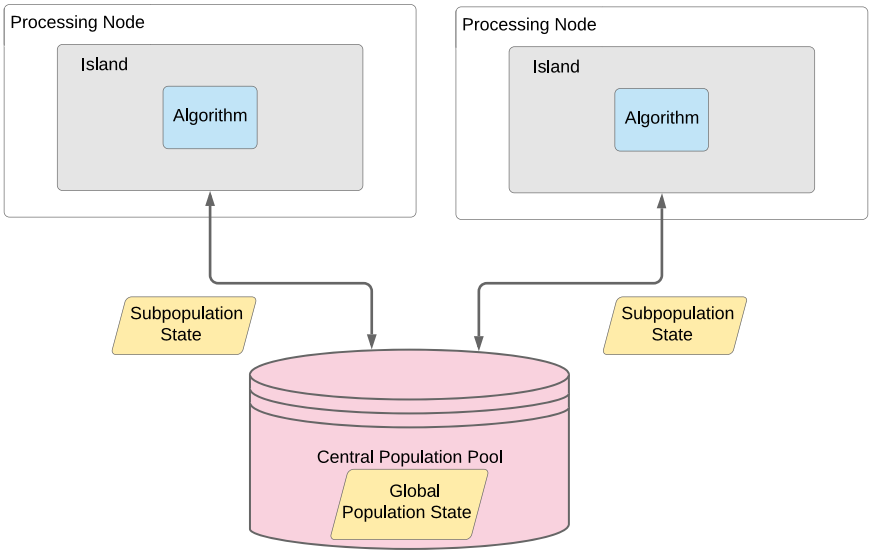


Fig. 5. Population state and dataflow between processing nodes of the EvoSpace algorithm.

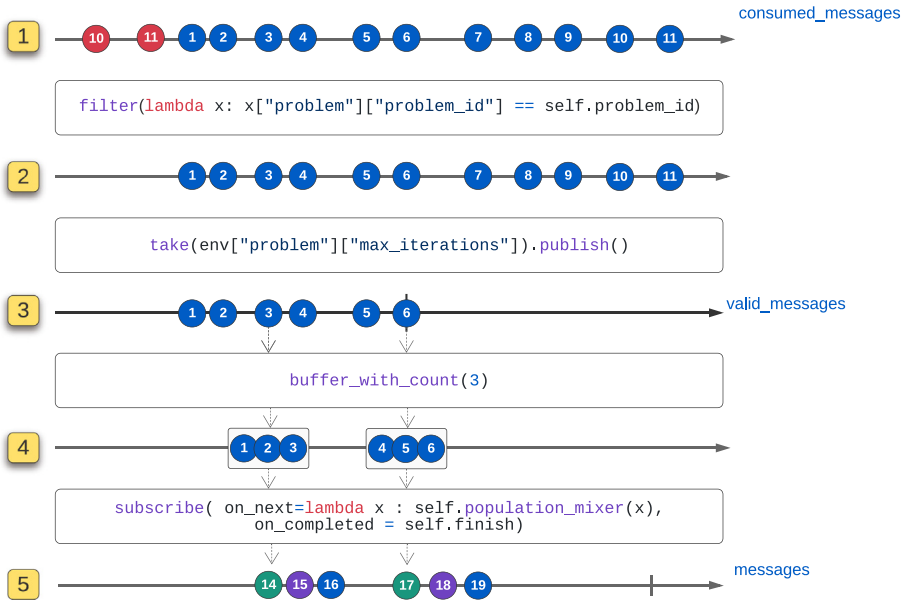


Fig. 6. Marble diagram for the Reactive Python implementation of the controller. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Algorithm 1 Migration

procedure CXBESTFROMEACH(pop_1, pop_2)
  $pop_1.sort()$
  $pop_2.sort()$
  $size \leftarrow min(len(pop_1), len(pop_2))$
  $cxpoint \leftarrow (size - 1)/2$
  $pop_1[cxpoint :] \leftarrow pop_2[: cxpoint + 2]$
  return pop_1
end procedure

5.3. Worker containers

Worker containers include a Python script called `main.py` running an infinite loop, continually trying to pull a new population to work on it. Once `main.py` receives a message containing configuration data, it creates a worker object responsible for initializing and running the specified stateless function (i.e., GA or PSO) with the population and configuration inside the message. Once the function returns the evolved population, the main script pushes the results to the `Output` queue and continues to execute the infinite loop. This process is shown in Fig. 7.

6. Workflow for reproducible experiment design in EvoSwarm

An essential goal of our EvoSwarm application is to allow researchers the description and orchestration of reproducible multi-population experiments. Following the application design described in the previous section, we now present the workflow of an experiment's entire life-cycle viewed from the perspective of the two roles users can play. On one side is the user role that designs and runs experiments and, on the other, a developer of algorithms. The scenario presented in Fig. 8 shows the workflow

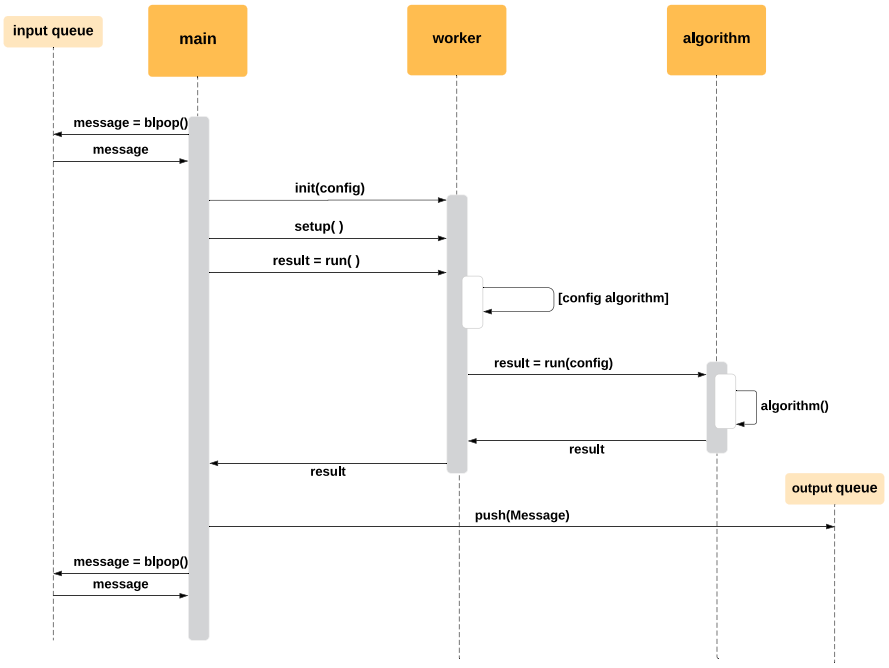


Fig. 7. Serverless function implementation details. Showing the worker algorithm in each container.

of an experiment in EvoSwarm and the two roles users can play. The first step is to develop the search strategy algorithms that are going to be used by the multi-population algorithm; algorithm developers do this. Then users are responsible for the configuration and deployment of the resources needed for the experiment. Once the application is running, users can start one or several experiments, monitor the execution, and analyze the results. Depending on the type of experiment executed, users could have additional needs. In this case, there are additional scripts for processing the results of experiments to be used with the COCO framework. In the next sections, we give more details about these roles and their workflows, followed by a proof-of-concept example of a reproducible experiment in EvoSwarm.

6.1. EvoSwarm from the developer point of view

Developers can add new algorithms as stateless functions in several ways and with two levels of integration. In the first level of integration, the algorithm execution is not dependent on an initial configuration created in the setup, and it only needs to take messages from the input queue and return the modified population to the output queue. At the second level, the algorithm receives a configuration from the initial setup, specifying the initial algorithm parameters. We show an example of the configuration file received by workers in Listing 1 in which we abbreviated the population key (the ellipsis is used instead). The message includes the population's current state and data about the problem, algorithm, and parameters.

Listing 1: Configuration message example

```
{"problem":{
  "name":"BBOB","instance": 1,"error":1e-8,"function":4,
  "dim":3,"search_space":[-5,5],
  "problem_id":"1515-4-1-3",
  "max_iterations": 20,
  "population":[
          [ -3.323452321611405,
            2.593007989886873,
            3.001112341167673],
            ...
          [ -2.22e343232411405,
            2.593007989886873,
            2.0011025031167673]
          ],
  "population_size": 100,
  "algorithm": "GA",
  "params":{
      "GA":{"crossover": { "type": "cxTwoPoint",
            "CXPB_RND": [0.2, 0.6], "CXPB": 0.26},
             "mutation": { "MUTPB_RND": [0.1, 0.3],
             "indpb": 0.05, "sigma": 0.5,
             "type":"mutGaussian","mu":0,"MUTPB":0.12},
             "selection": {
               "type": "tools.selTournament",
               "tournsize": 2 },
               "iterations": 50},
       "PSO": {"Vmax":5,"wMax":0.9,"wMin":0.2,
               "c1":2,"c2":2,"iterations":50 }
    },
  "experiment":{"type":"benchmark","experiment_id": 1515}
  }
}
```

In the first level of integration, developers can implement a new stateless Docker container following the description in the previous Section 5.3. The worker container executes a daemon that continuously pulls messages from the Redis host's input queue specified as an environment variable REDIS_HOST. If the new population-based algorithm is written in Python, developers can base the new implementation on the provided worker container image, adding a new algorithm function and worker and modifying the `main.py` file. The worker must include a constructor with the configuration dictionary (see Listing 1) as the only parameter and call the new function with the required parameters, unpacking them from the configuration dictionary if necessary. All the code of the EvoSwarm Application is published with an open-source MIT license.

If developers use an existing library for population-based algorithms, they need to change the standard model of execution, which creates a random population because, in this case, the algorithm needs to start with the population provided as a parameter. Depending on the evolutionary (or other) algorithm library used, it may be required to extract the population from the JSON message and replace the population object created by

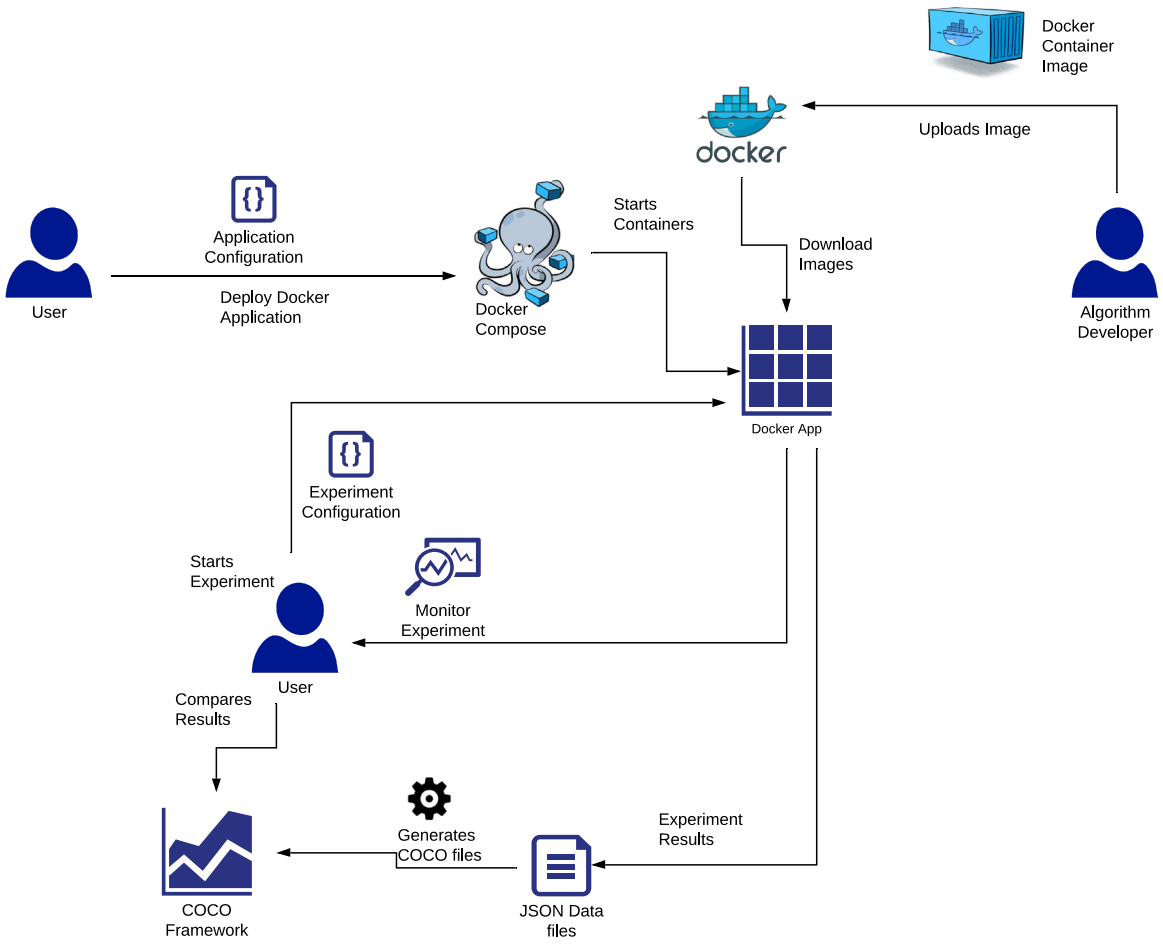


Fig. 8. EvoSwarm workflow.

the default library implementation. For example, the DEAP library we used for the GA algorithm creates the type of the population at runtime (i.e., bit, float), so we have to replace the population once DEAP creates it. In the case of EvoloPy, the PSO method directly uses a list of floats for storing the swarm, so we just added a parameter to receive the population as a list of floats. Both libraries store information about each iteration, so we just read it and put it in the format required. Again, the population's state must be encoded as a JSON message to be returned as a result, with additional data as depicted in Listing 2. The output message contains the worker container's id together with records giving information about each iteration of the algorithm, including the best solution found. The controller uses this information to track the algorithm's execution and keep a log in the Redis container for posterior analyses.

Listing 2: Fragment of an output message

```
{
"time_stamp": 1588455316.698875,
"evals": [
  {
    "gen_num": 0, "best_fitness": -259.96925945939404,
    "best_solution": [ -4.9446848321611405,
                       2.593007989886873,
                       2.0011025031167673],
    "num_of_evals": 17
  },
  {
    "gen_num": 49, "best_fitness": -419.47555780783347,
    "best_solution": [ -0.9165294450108963,
                       2.291966676472422,
                       1.8527006568795485],
    "num_of_evals": 28
  }
],
"worker_id": "4f5eb12e-8278-40de-b0f3-4ac18171566d",
"message_counter": 1,
"message_id": "cbcd2e82-e232-4818-8aec-2cd072c7cfe9",
"best_score": -419.47555780783347
}
```

For the secondary level of integration, developers need to add code to the setup container to specify additional configuration options. For instance, in the current configuration message (see Listing 1), workers executing a GA initialize the mutation and crossover probabilities randomly from a specific range of values. Users specify this range on the experiment configuration. Migration between populations is independent of the type of algorithm, as the controller treats messages as the same type of objects. However, developers could also change the controller to specify new operation rules, for instance, to change the type of migration depending on the algorithm. For example, in swarm-based algorithms, migration between populations (swarms) can follow a topology, indicating valid connections between them (i.e., ring or hypercube). The controller can apply these rules selectively only to swarm-based algorithms. In general, these configuration options will be algorithm and library-specific.

On either level of integration and after making the changes outlined before, algorithm developers can upload the container definition or images directly to a public repository like DockerHub.

Another option is to upload the container image definition (Dockerfile) to GitHub and define a trigger to re-upload the image

to the registry after an update. Having images publicly available can contribute to the reproducibility of experiments. Moreover, having more algorithms available can benefit researchers trying to compare against other algorithms or increase their algorithms' diversity by adding more search methods. Finally, in some cloud services, having hosted images is a requirement, as it is the case of Amazon ECS.

Moreover, public repositories can document, share, and track different versions of both images and image definitions by using tags. This infrastructure supports this paper's objective, which is to contribute to the reproducibility of experiments and platform independence.

6.2. EvoSwarm from the user's point of view

The user role is responsible for editing the `docker-compose` configuration files that reference the images created by developers. To deploy the application, he or she needs to edit the `docker-compose.yml` file to specify the type and number of containers required for the experiments. The `docker-compose` application is then responsible for starting all the containers and downloading new versions if necessary by executing the command `docker-compose up` inside the EvoSwarm directory.

After starting the application, users can push several Experiment Definition files and monitor the current execution using the PC console. After an experiment is finished, the user must execute another script that takes as argument the experiment id to generate a collection of files containing all the data generated by the experiment, again in a JSON format. An example of an experiment definition is shown in Listing 3. We give more details about the data contained in the file in the next section. Users can use these files to analyze and plot experiment data. We include Python scripts in the repository to plot the running times for the experiments and all other charts used in this paper; there is also a script to generate the files needed by the COCO framework [56], which can generate standard comparisons against other methods.

Listing 3: Experiment definition example

```
{
"DIM_CONFIGURATION" :
 {
 "10":{"GEN":50,"POP_SIZE": 70,"MAX_ITER":30,"MESSAGES":10},
 "20":{"GEN":66,"POP_SIZE":100,"MAX_ITER":30,"MESSAGES":10},
 },
"GA_WORKER_RATIO" : 0.5,
"FUNCTIONS": [4],
"DIMENSIONS" : [10, 20],
"CXPB_RND": [0.2, 0.6],
"MUTPB_RND": [0.1, 0.3]
}
```

6.3. Deploying to a cloud provider

EvoSwarm describes the deployment of loosely coupled processes in a consistent and portable way. The term cloud-native describes these container-based environments because we can deploy them on an infrastructure that abstracts the underlying compute, storage, and networking primitives. This infrastructure does not need to be a cloud provider; it could be a developer's laptop, a single server, or a production environment. This section gives specific details about how we deployed EvoSwarm in a cloud setting for running the experiments within this paper. Running the cloud-native application in the cloud has the obvious benefit of a multi-instance deployment. We can run containers in several computing nodes at the same time, not only in a single machine as it is the case when using docker-compose.

When we deploy a cloud-native application to a cloud provider, there are, among others, two options to choose from:

A Kubernetes cluster provides auto-scaling and management of containerized applications, and it supports other container technologies, but it needs additional configuration for provisioning, and its cost might be higher. All major cloud-providers have Kubernetes services. This option was used by S. Zhao et al. [57] to run a multi-island algorithm, and Dziurzanski et al. [51] later refined it.

A container orchestration service : These services are compatible with `docker-compose` files for configuration and provide transparent provisioning. Examples of these services are Amazon Elastic Container Service, Red Hat Openshift (which is also open source), and Marantis Docker Enterprise using Docker Swarm. Salza et al. [18] tested a variant of this option by using a CoreOs cluster.

We chose the second option, a container orchestration service. We deployed the EvoSwarm application in Amazon Elastic Container Service (ECS) since it needs fewer configuration changes (from deploying it locally using `docker-compose`) and it has a lower cost than the Kubernetes alternative. This does not imply any loss of generality; first, because it is `docker-compose` compatible, and it can be deployed in approximately the same way to any of the services, mentioned above, compatible with it. Since `docker-compose` is open source, it can be ported to any computing node, and can thus be deployed to any other cloud service like Azure or Google Cloud Platform. At any rate, please note that there are several levels of infrastructure programming here. At the lowest level, we are using docker containers, which are compatible across a myriad of cloud providers in one way or another. At the next level, we are using a container composition technology that is also standard, `docker compose`, once again compatible across any platform that can host that kind of service. There is, however, a final, platform-specific, infrastructure definition that will have to vary across any public or private cloud platform that we are using, and that will tap the cloud provider API. These changes, however, are usually minimal, and can also be done through the CLI or using the web console.

For instance, we can define an ECS cluster directly on the AWS Elastic Cloud VMs or through the AWS Fargate service. Fargate is a serverless compute engine for containers that provides automatic provision and management of AWS ECS resources. With this service, calculating the cost of execution is simplified because users only specify the computing units and memory for the application, instead of every component. However, AWS Fargate is not suited for an experimental setting because many of the details are out of user control. For this reason, we chose the alternative of using a cluster of EC2 containers. The drawback is that we need to manually provision the EC2 containers, networking, and group permissions for each experimental setup; for instance, when going from 2 to 8 workers, we need to increase the number of EC2 instances to meet the required amount of CPU units.

Multicontainer applications are deployed to ECS as a series of tasks *tasks*, and a single cluster can run multiple tasks in parallel. Using the ECS CLI, we can provision a cluster with a configuration file specifying the computing resources needed. We can also specify a task's configuration by using a task definition file (see Listing 4) in YAML format. The main attributes are the cpu_shares indicating the CPU units assigned, in this case, 2048, and the network configuration; in this case, we have two private subnets and a security group in this configuration. When deploying to EC2 containers, these requirements need to be supported by the currently available cluster. For instance, we need to have all the CPU units required by all tasks because if they are not available, the task initiation will fail.

The complete task configuration is generated on-the-fly by the ecs compose command that uses a docker-compose.yml file to

Table 1
Details of EC2 instance types used in our deployment.

Name	Memory	vCPUs	Storage	Cost
R5AD Large	16.0 GiB	2 vCPUs	75 GiB	$0.131000 hourly
C5 High-CPU Large	4.0 GiB	2 vCPUs	EBS Only	$0.102000 hourly
C5 High-CPU XLarge	8.0 GiB	4 vCPUs	EBS Only	$0.154000 hourly

generate the configuration. For example, to deploy the controller, setup, and Redis containers, we used the file in Listing 5.

Each container is defined as a service (a *task* in AWS terms), with the following attributes: image, that reference the container images we have published in Docker hub,[3] open ports, a command to be executed at startup, configuration for the AWS logging service, and finally, environment variables used by code running inside containers.

Fig. 9 shows the basic EC2 configuration used for the deployment of experiments. We selected three types of EC2 instances according to our needs; the setup, controller, and redis containers do not demand many resources and we do not need to scale them as the number of parallel workers is increased. They run in a single C5 High-CPU large instance. With the available 2048 CPU shares divided as follows: setup (128), controller (896), redis (1024). With this assignment, the redis container gets a dedicated vCPU because it is the most demanding task. We assigned 2048 units to each worker container and used a more capable C5 High-CPU XLarge instance, with four vCPUs and 4096 CPU shares. With this configuration, each worker gets a pair of vCPUs. We scaled the number of instances as needed, using one instance for every two worker containers.

Because all these instances run in a private subnet, we also run a Bastion instance, a specially protected network node, in a public subnet. From this instance, we send the commands to execute the experiments and later collect the results. We connected to this instance through an ssh session from our workstation. This is an R5AD Large instance, selected with 75GiB of storage, needed for keeping the logs, and more memory (16 GiB) for processing the logs in memory. The details of these EC2 instances and their cost at the time of writing are shown in Table 1.

Listing 4: ecs-params.yml

```
version: 1
task_definition:
  ecs_network_mode: awsvpc
  services:
    worker:
      cpu_shares: 2048

run_params:
  network_configuration:
    awsvpc_configuration:
      subnets:
        - "subnet-0ca2a542bbd74239e"
        - "subnet-0dfd367b5ca531b07"
      security_groups:
        - "sg-0681c18469874b391"
```

Listing 5: docker-compose.yml for setup, controller and redis containers

```
version: '3'
services:
  redis:
    image: "redis:alpine"
    ports:
      - "6379:6379"
    logging:
      driver: awslogs
      options:
        awslogs-region: us-west-2
        awslogs-group: docker-logs
        awslogs-stream-prefix: redis

  setup:
    image: mariosky/setup
    command: python experiment.py
    environment:
      PYTHONUNBUFFERED: 1
      REDIS_HOST: localhost
    logging:
      driver: awslogs
      options:
        awslogs-region: us-west-2
        awslogs-group: docker-logs
        awslogs-stream-prefix: setup

  controller:
    image: mariosky/controller
    command: python controller.py
    environment:
        PYTHONUNBUFFERED: 1
        REDIS_HOST: localhost
    logging:
      driver: awslogs
      options:
        awslogs-region: us-west-2
        awslogs-group: docker-logs
        awslogs-stream-prefix: controller
```

The complete configuration files are available at public the GitHub repository,[4] and they are used in the next section to carry an extensive evaluation of the framework. Again, each configuration describes the deployment of containers in a consistent and portable way. It was used in each phase of the life cycle, from development through testing to production independently on the kind of infrastructure.

7. Experimental study

In the previous section, we have described an option for an EvoSwarm deployment that can be used by the public to create reproductive research in the cloud. In this section, we are using the same framework to learn if the proposed solution can efficiently improve population-based optimization algorithms' scalability and performance. Hence, in the following sections, we provide answers to the following questions:

- Can we improve the algorithm's execution time by adding populations and serverless functions (workers) to the system and, in particular, what is its effect on the system's scalability?
- Does having a multi-population enabled platform, with heterogeneous populations and the support for mixing search strategies, increase the search's performance by needing fewer function evaluations than a homogeneous setting?

Section 7.2 is devoted to answering the first question, and Section 7.3 to the second. Next, we describe the general experimental setup designed to answer both questions.

[3] They are shown as different repositories for user mariosky here: https://hub.docker.com/u/mariosky.

[4] https://github.com/mariosky/EvoSwarm/tree/master/aws-EC2.

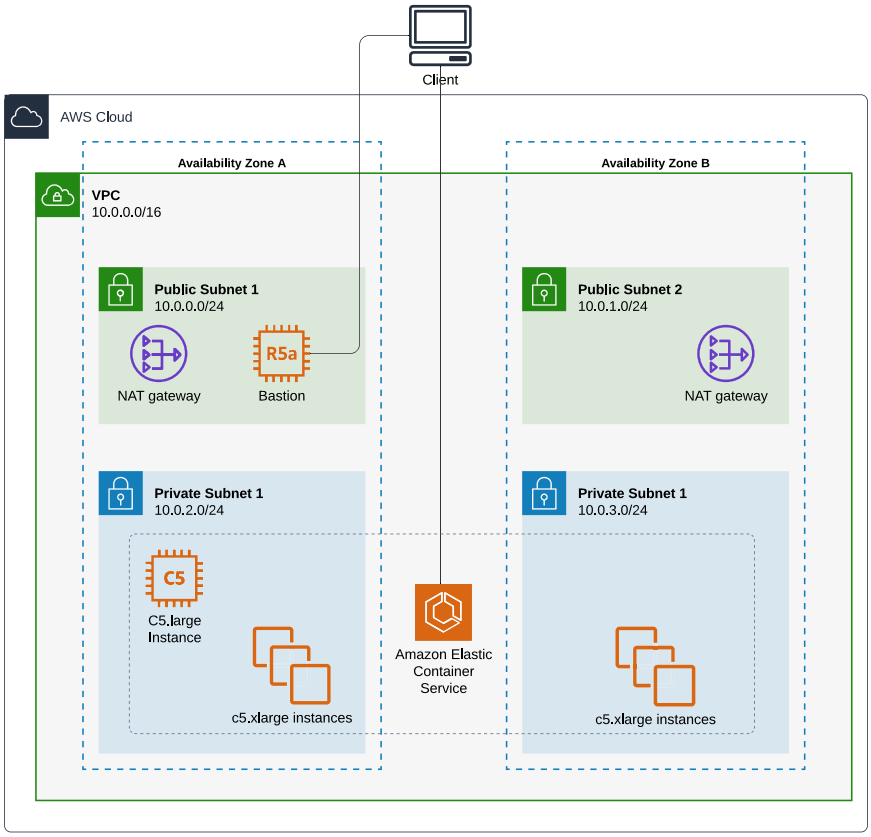


Fig. 9. AWS EC2 configuration for the deployment of EvoSwarm containers.

7.1. General experimental setup

To validate these questions, we used benchmark functions from the Continuous Noiseless BBOB testbed, which is part of the Comparing Continuous Optimizers (COCO) framework [56]. The testbed includes 24 real-parameter, single-objective benchmark functions, and the capability to provide additional instances of each function. Each instance of a function has a different optimal value. The standard benchmark runs 15 instances per function over 2, 3, 5, 10, 20, and 40 dimensions. The maximum number of Function Evaluations (FEs) changes according to the dimension (D), and it is determined by the expression $10^5 \cdot D$. As an example, if we have $D = 2$, the maximum number of FEs is 200,000.

The COCO framework offers several tools to compare algorithms' performance, generating data sets, tables, and reports for an experiment. There is a repository[5] of more than 200 results for the noiseless BBOB testbed, collected from BBOB workshops and special sessions between the years 2009 and 2019. The EvoSwarm application includes an adapted version of the noiseless BBOB testbed, compatible with the scripts of the framework, to compare with other algorithms stored in the repository.

To test the heterogeneous multi-population capabilities, we compare the performance between a homogeneous and an ensemble of multi-populations, using Genetic Algorithms (GAs) and Particle Swarm Optimization (PSO). We used the DEAP library [58], for the GA implementation. For the PSO algorithm, the EvoloPy library [59]. Both Python libraries are open-source. In EvoSwarm, we implemented these two algorithms as stateless functions.

Next, we show each algorithm's parameters, with Table 2 for the GA and Table 3 for the PSO. We obtained these parameters following the same method as in previous work [60]. To obtain the parameters, we tested first on the Rastrigin separable function with five dimensions. After about fifteen experiments, the algorithm achieved the most challenging targets for this particular function. We tested again with functions one to three, and after obtaining favorable results, we set the PSO and GA parameters to these values. In the GA, we randomly set the mutation and crossover probabilities to have more heterogeneous workers; for these parameters, we specify the range of values in Table 2. We did not change these parameters during the experiments and only modified the number of populations, population size, and generations. This strategy has been applied successfully for increasing the performance of multi-population algorithms without requiring the setting of initial parameters [61].

The experiments can be easily reproduced by using the configuration files described in Section 6.2. As a first step, we can deploy the docker application using a `docker-compose.yml` file, as

Table 2
DEAP GA EvoWorker parameters.

Selection	Tournament size = 12
Mutation	Gaussian $\mu = 0.0$, $\sigma = 0.5$, indbp = 0.05
Mutation probability	[0.1,0.3]
Crossover	Two point
Crossover probability	[0.2,0.6]

Table 3
EvoloPy PSO parameters.

V_{max}	6
W_{max}	0.9
W_{min}	0.2
C_1	2
C_2	2

[5] https://coco.gforge.inria.fr/doku.php?id=algorithms-bbob.

Table 4
Experiment configuration example.

Parameter	Type	Example
Worker containers	`int`	`8`
GA-PSO ratio	`decimal`	`0.5`
Benchmark functions	`list`	`[1, 2, 3, 4, 5]`
Instances	`integer`	`15`
Dimensions	`list`	`[10, 20, 40]`
Crossover probability range	`list`	`[0.2, 0.6]`
Mutation probability range	`list`	`[0.1, 0.3]`

Dimensions

Dimension	Generations	Population size	Populations	Iterations
10	50	140	5	30
20	66	200	5	30

Table 5
Parameters used in the experiments.

Populations	Population size	Generations	Maximum iterations
10 Dimension			
5	70	50	60
10	30	15	200
20	15	10	400
20 Dimension			
5	100	66	60
10	60	15	200
20	30	8	400

described in Section 6.2 for a local test, or by following the steps for a cloud setting as described in Section 6.3. Then a JSON file containing the configuration parameters of the experiment has to be provided. Table 4 shows an example of these parameters. The *GA-PSO Ratio* parameter indicates the proportion of populations that will use the GA algorithm. In the example, with a value of 0.50, there will be about the same proportion of GA and PSO populations. If we specify a value of 0, this will give us an algorithm with only PSO populations, and finally, a value of 1 means that every population will run the GA algorithm.

Next, we specify a list indicating which of the 24 benchmark functions we will test. In the example, the experiment will use the first five functions. The *Instances* parameter indicates how many instances of each function we will test. *Instances* have a default value of 15. We used this value because it is the standard for the BBOB benchmark [56]. In the *Dimensions* parameter, we define a list of the dimensions that will be tested, and we must select additional parameters for each dimension. According to the maximum number of FEs we mentioned earlier, for each dimension, we define the number of populations that will be generated in the setup, and for each population, the number of generations, and population size. Finally, the *Iterations* parameter indicates how many complete loops, that is, the maximum number of messages taken from the queue. All these parameters give us the maximum number of FEs we will have as budget. For instance, for $D = 2$, the maximum number of FEs is 200,000, which is the same as $40 * 50 * 10 * 10$.

We have deployed the EvoSwarm application in Amazon Elastic Container Service as described in Section 6.3, each EC2 instance was an Amazon Linux 2 AMI with ECS Agent 1.42.0 and Docker version 19.03.6-ce. Additionally, the BBOB experiments were performed with COCO [56] version bbob.v15.03 in Python, and the plots were produced with version 2.3.2.

7.2. Measuring scalability

As we mentioned before, in Section 4, reactive systems can scale by adding additional copies of serverless functions. In our case, we can start additional worker containers to have the same effect. Other authors, like Salza et al. [18], have used a similar architecture, which implies that worker nodes need to have at least a certain level of complexity, in terms of execution time, to scale on multiple nodes, tending to linear scalability. In that paper, Salza et al. tested scalability by simulating the nodes' work by using a sleep function and messages of different sizes. In a population-based algorithm like EvoSwarm, several parameters can increase or decrease the workload of workers:

- The number of populations: if there are more workers than populations, workers must wait for work to arrive. If there are too many populations, they could be standing in the queue for more time.
- The size of populations and the number of generations: These parameters naturally increase the execution time, because they mean more FEs.
- The complexity of the algorithm: For instance, in our case, the PSO implementation has a lower execution time than the GA.

From this description, we can deduce that we give more importance to the number of populations since they are the basic unit of work, the message, that can be executed in parallel. Together with each population's size and the number of generations, this parameter determines the maximum number of function evaluations for a single iteration. According to this, we do not have control over each algorithm's complexity, but future analyses could focus on balancing populations and the number of worker nodes.

In this experiment, we evaluate how the number of populations and workers are related to the system's scalability. In the next section, we describe the experimental setup. A discussion of the results follows, concerning the speed-up in terms of evaluations per second (Section 7.2.2) and execution time (Section 7.2.3).

7.2.1. Setup

To answer the first research question, we need to evaluate the effect of the number of populations and the number of workers on the system's scalability. To achieve this, we propose an experiment for which we have selected f_4 (Skew Rastrigin-Bueche separable) from the BBOB testbed, over 10 and 20 dimensions. This function has been used because it is computationally demanding, and in higher dimensions, it has been difficult for PSO [62] and GAs [63] to solve with the required number of FEs. As we are only comparing in terms of scalability, the results from a single function can be better understood since fewer factors are involved. Following the procedure described in Section 7, we executed two sets of experiments with five, ten, and twenty initial populations, repeating each experiment 30 times using 1, 2, 4, 8, and 16 workers. We kept the same maximum number of FEs, changing the relevant parameters for this. See Table 5 for the complete list.

7.2.2. Results regarding speed-up

When comparing the results in this section, we performed the Wilcoxon Signed-Rank test, with a significance level of 5%. We indicate with an asterisk those results that are not significant throughout the tables of this section. To measure the speed-up obtained by the addition of workers, we will use the number of evaluations per second instead of the time required to finish an instance of a problem. This is because in some cases reaching the desired target can require less work because we are using a stochastic search. We are going to measure the total increase of work done in a second by adding more workers.

The baseline for the comparison will be the algorithm with a single worker, and the minimum amount of work available will be five populations. This number is selected because the controller

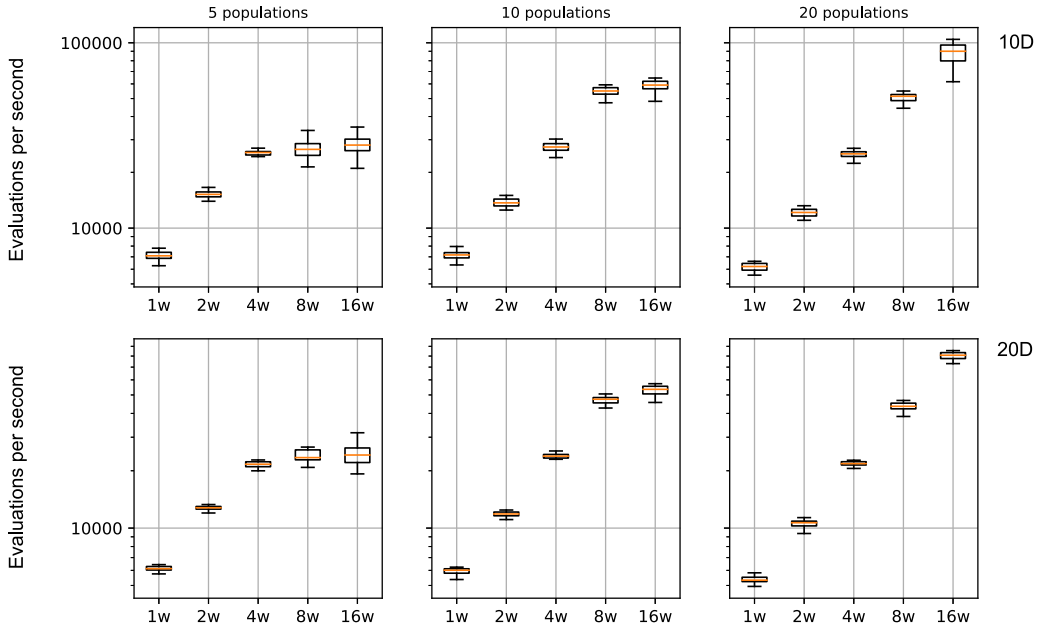


Fig. 10. Boxplot of the number of evaluations per second, for 30 instances of function f_4, with a different number of workers and populations. The y-axis is in log scale.

Table 6
Speed-up by worker and dimension, taking one worker as the baseline. Speed-up is obtained by dividing the baseline's evaluation ratio by the ratio of the increased number of workers.

Dimension	2W	4W	8W	16W
5 Populations				
10	**2.15**	3.61	3.75	3.96
20	**2.09**	3.53	3.83	3.95
10 Populations				
10	1.91	3.82	7.67	8.25
20	1.97	3.97	7.90	8.90
20 Populations				
10	1.96	**4.05**	**8.29**	**14.50**
20	2.00	**4.09**	**8.18**	**15.16**

needs at least three populations to migrate individuals between them, and we add two more to keep even a single worker busy all the time. Fig. 10 shows the number of function evaluations per second, achieved by a certain number of workers, for ten and twenty dimensions of f_4.

As expected, there is a linear speed-up when we add more workers within the same number of populations. The speed-up is marginal in those cases where there are fewer populations than workers. This is because, at any given time, there are at least two populations in the message queue, leaving some of the workers waiting for populations to be available in the queue.

Table 6 shows the speed-up obtained against the baseline of one worker. We achieved the maximum speed-up when using four workers or more with twenty populations. On the other hand, the maximum speed-up when we consider the cost was with eight workers. The best speed-ups for each number of workers are shown in bold in the table.

Table 7 shows the speed-up obtained taking as the baseline the previous configuration, such as increasing from four to eight workers, and the p-value for the Wilcoxon Signed-Rank test. There was an increment in all cases. Again, the best effect is achieved when the number of populations is higher than the number of workers. Configurations following this basic rule doubled the speed or where near that. Using five populations, we obtained the best and worst speed-up depending on the number of workers (shown in underline and bold, respectively).

Table 7
Speed-up achieved with increasing the number of workers for each dimension showing the p-value for the Wilcoxon test. Best and Worst are highlighted (bold and underlined).

Populations	Dimension	Increment		Speed-up	p-value
5	10	1w	2w	2.15	1.51E−11
	10	2w	4w	1.68	1.51E−11
	10	4w	8w	1.04	0.004942
	10	8w	16w	1.05	0.072660
	20	1w	2w	**2.09**	1.51E−11
	20	2w	4w	1.69	1.51E−11
	20	4w	8w	1.08	5.97E−07
	20	8w	16w	<u>1.03</u>	0.214482
10	10	1w	2w	1.91	1.51E−11
	10	2w	4w	2.00	1.51E−11
	10	4w	8w	2.01	1.51E−11
	10	8w	16w	1.08	1.39E−05
	20	1w	2w	1.97	1.51E−11
	20	2w	4w	2.01	1.51E−11
	20	4w	8w	1.99	1.51E−11
	20	8w	16w	1.13	4.24E−09
20	10	1w	2w	1.96	1.51E−11
	10	2w	4w	2.07	1.51E−11
	10	4w	8w	2.05	1.51E−11
	10	8w	16w	1.75	1.51E−11
	20	1w	2w	2.00	1.51E−11
	20	2w	4w	2.04	1.51E−11
	20	4w	8w	2.00	1.51E−11
	20	8w	16w	1.85	1.51E−11

7.2.3. Results regarding execution time

In the previous section, we have shown that as we increase the number of populations and workers, we increase the number of evaluations per second in the same proportion. However, it is vital to notice that we can also affect the number of evaluations we need to execute to find a solution. This reduction means that the multi-population based algorithm can perform better when we increase the number of populations, or in some cases, even workers, as the experiments show. If we increase the evaluations per second and, at the same time, decrease the number

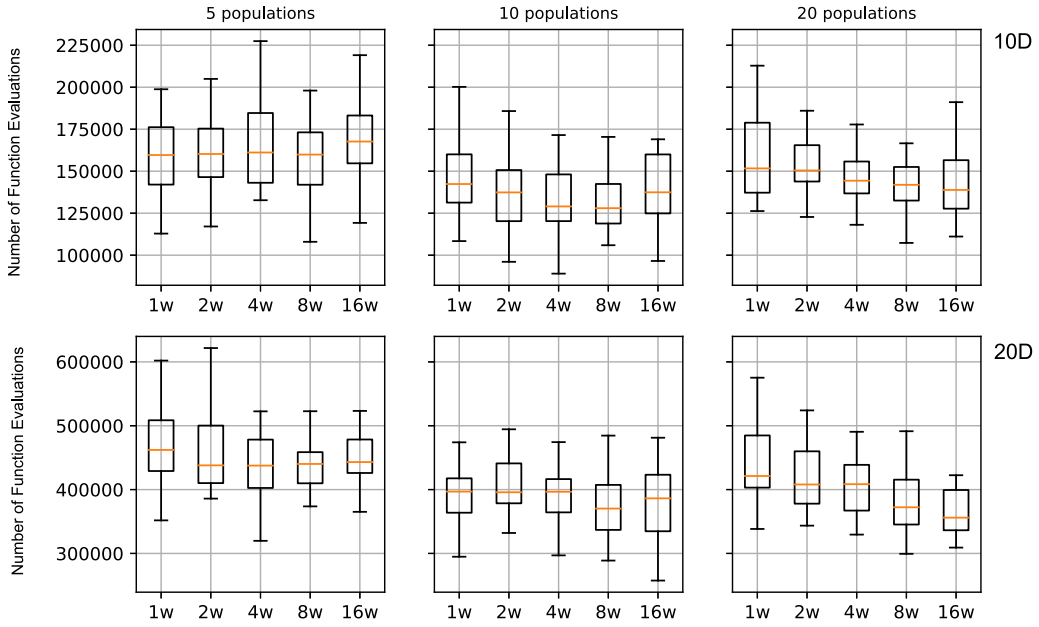


Fig. 11. Boxplot of the number of function evaluations required to reach the target, for 30 instances of function f_4, with a different number of workers and population sizes.

Table 8
Median of the number of evaluations required to finish an instance of the function f_4 in 30 runs with a different number of workers and populations.

Dimension	Workers	Number of populations		
		5	10	20
10	1w	159,625	**142,368***	151,680*
	2w	160,258	**137,385**	150,445*
	4w	161,144	**129,031**	144,354
	8w	159,893	**127,987**	141,896
	16w	167,678	**137,446**	138,852
20	1w	462,069	**396,841**	421,249*
	2w	437,952	**395,814**	407,984
	4w	437,563	**396,587**	408,461
	8w	440,130	**370,264**	372,265
	16w	443,108	386,272	**355,999**

Table 9
The median of the time required to finish an instance of the function f_4, 30 runs with a different number of workers and populations.

Dimension	Workers	5 Populations Median of time	10 Populations Median of time	20 Populations Median of time
10	1w	22.24	20.48	24.55
	2w	10.31	**10.00** (2.05)	12.79
	4w	6.23	**4.75** (2.10)	5.70
	8w	5.86	**2.33** (2.03)	2.74
	16w	5.96*	2.33*	1.59
20	1w	73.87	65.18	80.31
	2w	34.18	33.46	38.87
	4w	19.55	**16.58** (2.02)	18.69
	8w	18.38	**7.77** (2.15)	8.66
	16w	18.36*	7.27	4.57

of evaluations needed, we can have a superlinear speed-up of the algorithm, if we measure the time required to complete an instance.

First, we will see how the configuration we selected can reduce the number of evaluations needed in an experiment. Fig. 11 shows the median (30 runs) of the number of evaluations needed to complete an instance of the function f_4. For every selection in the number of populations, we could expect to have about the same number of FEs needed regardless of the number of workers. However, we can see that for some configurations, i.e., 10D and ten populations, eight workers will need fewer FEs than one ($p = 0.012$). Also, for 20D and twenty populations, there are differences between one and four workers ($p = 0.08$) and between four and sixteen ($p = 0.01$). These results indicate that there are benefits in the potential change of order in the message queue resulting from work parallelization.

In Table 8 we try to acquire more insights on how this works. When we increase the number of populations from five to ten, or twenty, the algorithm needs fewer FEs in both cases. The reduction is more pronounced when using ten populations, needing fewer FEs in all but a single case (shown in boldface). In some cases, marked with an asterisk, there is no statistical difference between the number of evaluations; this happens when using only one or two workers. The configurations that required less FEs have an underline in the table.

The reduction in the number of FEs impacts when we measure the speed up concerning the time needed to complete an experiment. In this case, we have measured the median of the seconds needed to complete an instance of the problem. Fig. 12 shows in each boxplot the median (30 runs) of the execution time achieved by 2, 4, 8, and 16 workers, to finish an instance of the function f_4, for dimensions 10 and 20, when using 5, 10, and 20 population sizes. We present each dimension in a row having a log scale on the y-axis to compensate for the dimension increments and scalability.

In Table 9 we can see that we have a supralinear speed-up in some cases; we highlighted these in boldface and indicated the speed-up value in parenthesis. The best overall times are underlined. As expected, increasing the number of workers for a fixed dimension and populations reduced the time required to find a solution, as long as the population number was higher than the number of workers. The values that had no statistical difference (in the Wilcoxon test) when increasing the number of workers have an asterisk.

7.3. Performance of heterogeneous populations in the BBOB benchmark

A multi-population based algorithm can decrease the execution time thanks to running the algorithm on several populations

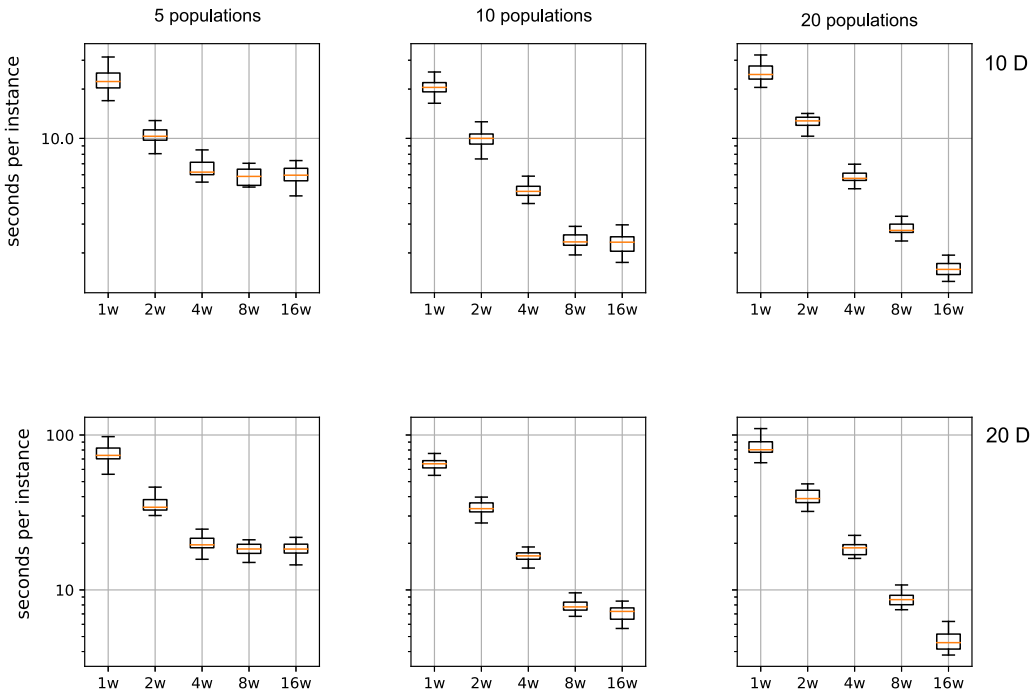


Fig. 12. Boxplot of the median time required for the execution of 30 instances of function f_4, in seconds, a different number of workers and populations sizes. The y-axis is shown in log scale.

simultaneously. However, if the populations are heterogeneous it might enhance the evolutionary search and avoid unneeded evaluations to a greater extent than homogeneous systems do; heterogeneous settings, if done right, increase the diversity of the whole population [64].

Nevertheless, this is a rule of thumb, and it will depend on the degree of heterogeneity and the algorithm itself. Some level of heterogeneity can be achieved by just changing the configuration parameters of each population, as in the previous experiment. However, in this case, we are interested in answering the second research question. We want to validate if we can increase the search's performance using heterogeneous populations and mixing search strategies.

Therefore, in this experiment, we compare a multi-population with GA and PSO only populations, versus an ensemble of GA and PSO algorithms. We have tested on the first five functions of the BBOB testbed; we have used ten populations and eight workers for the experiment and the same parameters as before.

To run this experiment, we just needed to edit a configuration file and start it up. The plots presented in the results are generated by the BBOB framework from the JSON log file for each run. There was no additional setup needed, proving how this framework can offer a reproducible way to run experiments and how easily it interfaces with other common tools in the EC ecosystem.

7.3.1. Experimental results

The results obtained by the experiments show (see Fig. 13) how the runtime scales with dimension to reach certain target values Δf; The color of the lines indicates the average runtime, for each target, reached at least once. A red circle without a number on top indicates the algorithm reached the 10^{-8} target on all instances for that dimension. If there is a number, it indicates how many times the algorithm has reached the target. Fixed values of targets $\Delta f = 10^k$ with k colors are given in the f_1 legend in the top row, k values are $[-8, -5, -3, -2, -1, 0, 1]$. Results from each configuration are presented in each column, GA, PSO, and GA&PSO populations.

We can see that the GA algorithm (on the leftmost column) has a hard time reaching even the 10^{-5} target in functions ($f_1 - f_4$) for all dimensions and has the worst performance in 20 and 40 dimensions. Finally, it performs better on f_5 reaching the 10^{-8} on all dimensions but with a higher runtime on lower dimensions.

On the other hand, the multi-population PSO (taking the column in the middle) has better performance; its results are spread over a smaller range, reaching all targets for functions f_1 and f_2 but with a slightly higher runtime than the multi-strategy configuration. It is not able to reach the 10^{-8} target on higher dimensions of f_3 and f_4. Moreover, the PSO multi-population has the best runtime for the lower dimensions of f_5.

Finally, the proposed PSO&GA hybrid algorithm has reached the 10^{-8} target value on all functions ($f_1 - f_5$). It scales nicely to higher dimensions, even on f_3 and f_4 (charts on the lower rows), usually considered difficult functions. This experiment also exemplifies the type of analyses that we can perform with the EvoSwarm application results by reusing the COCO framework. We also proved that multi-population algorithms could be compared under the same conditions, including single and multi-algorithm versions, and results show that the latter has the best performance.

8. Conclusions

This paper has presented the EvoSwarm framework for the reproducible execution of experiments with an event-based, cloud-native architecture suitable for hosting multi-population, multi-algorithm optimization methods. The framework is based on industry-standard development tools such as Docker and Docker Compose, making it easy for scientists and practitioners to develop and deploy multi-algorithm distributed experiments. The model implemented by the framework is based on the asynchronous exchange of messages between stateless functions that react to a continuous stream of data; populations are the messages that flow in this stream. We show the differences of the message-driven architecture we presented against four other proposals for multi-population algorithms found in the literature

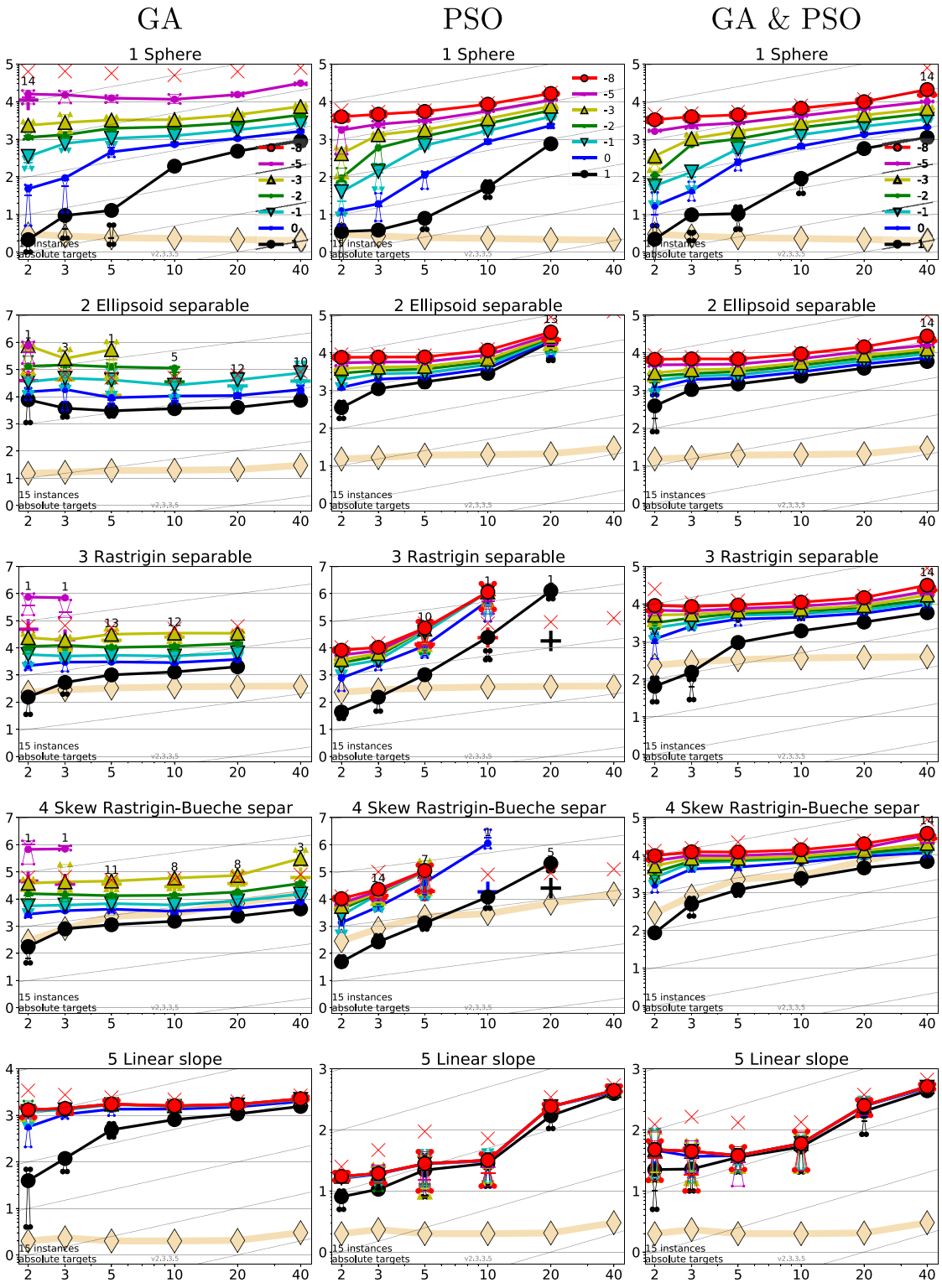


Fig. 13. Scaling of expected runtime (RT defined as the number of function evaluations #FE) to reach a target for each dimension. Target values $\Delta f = 10, 1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-5}, 10^{-8}$ (the exponent and colors are given in the legend of f_1). Lines are the average RT; Cross (+): median RT of successful runs to reach the most difficult target that was reached at least once; Cross (×): maximum #FE in any trial. Values are divided by dimension and plotted as log_{10}. The light thick line with diamonds indicates the theoretical best algorithm from BBOB 2009 for the most difficult target. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

and highlight the advantages of conducting asynchronous multi-population algorithms.

We have proposed a new methodology for the reproducible deployment and execution of multiple experiments by specifying the infrastructure as part of an experiment definition in both local and cloud environments. We presented the workflow of an experiment's entire life-cycle viewed from the perspective of developers and researchers. We have tested the deployment of an experiment on Amazon's Elastic Container Service and proved that as a cloud-native, container-based application, EvoSwarm

could be deployed locally or in the cloud by specifying the configuration of container images and resources in an orchestration service.

We have performed an empirical evaluation of the same framework to learn if the proposed solution can efficiently improve population-based optimization algorithms' scalability and performance.

First, we focused on knowing how scaling is related to the number of populations and workers in the system. This experiment concludes that there is an interesting interplay between them, with the most benefit obtained with a number of populations slightly above the number of workers, which brings performance gains in terms of execution speed; these experiments take less time since evaluations are performed, as part of its corresponding algorithm, simultaneously by different workers. Also, we found that for lower dimensions, there is an advantage in using fewer populations.

A second experiment centered on evaluating the algorithmic benefits (experiments are taking less time because they need fewer evaluations) of distributed multi-algorithm algorithms such as the ones implemented in this kind of framework. We compared the homogeneous and an ensemble of multi-populations, using Genetic Algorithms (GAs) and Particle Swarm Optimization (PSO), using the COCO benchmark framework to visualize and define optimization problems. We found that these algorithms' setup has a significant influence on their results since diversity is essential for their performance. The hybrid method using both algorithms achieved better results and, in most cases, needed fewer evaluations.

We conclude that superlinear improvements are achievable by selecting an appropriate configuration consisting of the number of populations, worker containers, and algorithm combinations. Further research should focus on learning more about how these parameters interact with each other. The BBOB benchmark results show that the selection and mixing of algorithms can lower the number of FEs needed to find a solution or find solutions that single algorithms could not find.

This work also opens new possibilities. From the design point of view, it would be interesting to use automatic scaling, instead of setting the number of workers by hand. This would need a certain amount of configuration, as we would be using container orchestration with auto-scaling capabilities such as Kubernetes or Docker Swarm. The current orchestration (Amazon ECS) also has auto-scaling capabilities, but additional configuration is needed; we also need to establish the auto-scaling of both containers and Amazon's EC2 instances. Instead of setting a fixed number of workers, only the maximum amount would need to be established, which could be done directly or by setting an evaluation budget.

From the algorithmic point of view, there are many possible lines of research. Mixing population-based algorithms with other kinds would be a possibility and using different instances of population-based algorithms, such as checking how Estimation of Distribution Algorithms would work together with evolutionary algorithms. EvoloPy also includes other population-based algorithms, which could be tested, trying to determine which sets of algorithms are a better match for each other. We will explore all this as future lines of work.

CRediT authorship contribution statement

Mario García Valdez: Conceptualization, Methodology, Software, Writing - original draft, Visualization. **Juan J. Merelo Guervós:** Conceptualization, Writing - review & editing, Preparation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

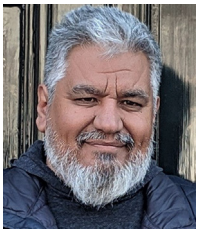
Acknowledgments

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

References

[1] X.-S. Yang, Nature-Inspired Optimization Algorithms, Elsevier, 2014.
[2] T. Back, Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford university press, 1996.
[3] J. Kennedy, Swarm intelligence, in: Handbook of Nature-Inspired and Innovative Computing, Springer, 2006, pp. 187–219.
[4] J.H. Holland, et al., Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, MIT press, 1992.
[5] A.E. Eiben, J.E. Smith, Genetic algorithms, in: Introduction to Evolutionary Computing, Springer, 2003, pp. 37–69.
[6] S. Mirjalili, S.M. Mirjalili, A. Lewis, Grey wolf optimizer, Adv. Eng. Softw. 69 (2014) 46–61.
[7] D. Karaboğa, S. Ökdem, A simple and global optimization algorithm for engineering problems: differential evolution algorithm, Turkish J. Electr. Eng. Comput. Sci. 12 (1) (2004) 53–60.
[8] M. Clerc, Particle Swarm Optimization, Vol. 93, John Wiley & Sons, 2010.
[9] M. Dorigo, G. Di Caro, Ant colony optimization: a new meta-heuristic, in: Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406), Vol. 2, IEEE, 1999, pp. 1470–1477.
[10] H. Mühlenbein, M. Gorges-Schleuter, O. Krämer, Evolution algorithms in combinatorial optimization, Parallel Comput. 7 (1) (1988) 65–85.
[11] M. Gorges-Schleuter, Explicit parallelism of genetic algorithms through population structures, in: International Conference on Parallel Problem Solving from Nature, Springer, 1990, pp. 150–159.
[12] P. Grosso, Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in Multilocus Model (Ph.D. thesis), University of Michigan, 1985.
[13] C. Li, T.T. Nguyen, M. Yang, S. Yang, S. Zeng, Multi-population methods in unconstrained continuous dynamic environments: The challenges, Inform. Sci. 296 (2015) 95–118.
[14] V. Coleman, The Deme Mode: An Asynchronous Genetic Algorithm, Tech. rep., University of Massachussets at Amherst, Dept. of Computer Science, uM-CS-1989-035, 1989.
[15] J.W. Baugh, S.V. Kumar, Asynchronous genetic algorithms for heterogeneous networks using coarse-grained dataflow, in: Genetic and Evolutionary Computation Conference, Springer, 2003, pp. 730–741.
[16] J.J. Merelo-Guervos, A. Mora, J. Cruz, A. Esparcia-Alcázar, C. Cotta, Scaling in distributed evolutionary algorithms with persistent population, in: Evolutionary Computation (CEC), 2012 IEEE Congress on, 2012, pp. 1–8, http://dx.doi.org/10.1109/CEC.2012.6256622.
[17] J. Merelo-Guervos, P. Castillo, J.L.J. Laredo, A. Mora Garcia, A. Prieto, Asynchronous distributed genetic algorithms with Javascript and JSON, in: Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on, June, 2008, pp. 1372–1379, http://dx.doi.org/10.1109/CEC.2008.4630973.
[18] P. Salza, F. Ferrucci, Speed up genetic algorithms in the cloud using software containers, Future Gener. Comput. Syst. 92 (2019) 276–289.
[19] J.J.M. Guervós, J.M. García-Valdez, Introducing an event-based architecture for concurrent and distributed evolutionary algorithms, in: International Conference on Parallel Problem Solving from Nature, Springer, 2018, pp. 399–410.
[20] R. Serrano, J. Tapia, O. Montiel, R. Sepúlveda, P. Melin, High performance parallel programming of a GA using multi-core technology, Stud. Comput. Intell. 154 (2008) 307–314, http://dx.doi.org/10.1007/978-3-540-70812-4_17.
[21] X. Lai, Y. Zhou, An adaptive parallel particle swarm optimization for numerical optimization problems, Neural Comput. Appl. 31 (10) (2019) 6449–6467.
[22] J.J. Merelo, J.L.J. Laredo, P.A. Castillo, J.-M. García-Valdez, S. Rojas-Galeano, Scaling in concurrent evolutionary algorithms, in: Workshop on Engineering Applications, Springer, 2019, pp. 16–27.
[23] Y. Tan, K. Ding, A survey on GPU-based implementation of swarm intelligence algorithms, IEEE Trans. Cybern. 46 (9) (2015) 2028–2041.

[24] J. Li, D. Wan, Z. Chi, X. Hu, An efficient fine-grained parallel particle swarm optimization method based on GPU-acceleration, Int. J. Innovative Comput. Inf. Control 3 (6) (2007) 1707–1714.
[25] P. Fazenda, J. McDermott, U.-M. O'Reilly, A library to run evolutionary algorithms in the cloud using MapReduce, Appl. Evol. Comput. (2012) 416–425.
[26] A. Munawar, M. Wahib, M. Munetomo, K. Akama, The design, usage, and performance of GridUFO: A grid based unified framework for optimization, Future Gener. Comput. Syst. 26 (4) (2010) 633–644.
[27] D.L. Gonzalez, F.F. de Vega, L. Trujillo, G. Olague, L. Araujo, P.A. Castillo, J.-J. Merelo-Guervós, K. Sharman, Increasing GP computing power for free via desktop grid computing and virtualization, in: PDP, 2009, pp. 419–423.
[28] N. Cole, T.J. Desell, D.L. Gonzalez, F.F. de Vega, M. Magdon-Ismail, H.J. Newberg, B.K. Szymanski, C.A. Varela, Evolutionary algorithms on volunteer computing platforms: The milkyway home project, in: Parallel and Distributed Computational Intelligence, Springer, 2010, pp. 63–90.
[29] J.-J. Merelo, M. García-Valdez, P.A. Castillo, P. García-Sánchez, P. Cuevas, N. Rico, Nodio, a javascript framework for volunteer-based evolutionary algorithms: first results, arXiv preprint arXiv:1601.01607.
[30] M. García-Valdez, L. Trujillo, J.-J. Merelo, F. Fernández de Vega, G. Olague, The EvoSpace model for pool-based evolutionary algorithms, J. Grid Comput. 13 (3) (2015) 329–349, http://dx.doi.org/10.1007/s10723-014-9319-2.
[31] R.M. Valenzuela, M.G. Valdez, Implementing pool-based evolutionary algorithm in Amazon Cloud Computing Services, in: Design of Intelligent Systems Based on Fuzzy Logic, Neural Networks and Nature-Inspired Optimization, Springer, 2015, pp. 347–355.
[32] D. Sherry, K. Veeramachaneni, J. McDermott, U.M. O'Reilly, Flex-GP: Genetic programming on the cloud, in: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), in: LNCS, vol. 7248, 2012, pp. 477–486, http://dx.doi.org/10.1007/978-3-642-29178-4_48.
[33] J. Thönes, Microservices, IEEE Softw. 32 (1) (2015) 116, http://dx.doi.org/10.1109/MS.2015.11.
[34] B. Varghese, R. Buyya, Next generation cloud computing: New trends and research directions, Future Gener. Comput. Syst. 79 (2018) 849–861.
[35] B. Varghese, R. Buyya, Next generation cloud computing: New trends and research directions, Future Gener. Comput. Syst. 79 (2018) 849–861, http://dx.doi.org/10.1016/j.future.2017.09.020, cited By 2.
[36] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, Cloud-native, event-based programming for mobile applications, in: Proceedings - International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016, 2016, pp. 287–288, http://dx.doi.org/10.1145/2897073.2897713.
[37] M. Malawski, A. Gajek, A. Zima, B. Balis, K. Figiela, Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions, Future Gener. Comput. Syst. 110 (2020) 502–514, http://dx.doi.org/10.1016/j.future.2017.10.029, URL http://www.sciencedirect.com/science/article/pii/S0167739X1730047X.
[38] J. Bonér, D. Farley, R. Kuhn, M. Thompson, The reactive manifesto, 2014, URL http://www.reactivemanifesto.org/.
[39] H. Ma, S. Shen, M. Yu, Z. Yang, M. Fei, H. Zhou, Multi-population techniques in nature inspired optimization algorithms : A comprehensive survey, Swarm Evol. Comput. 44 (2017) (2019) 365–387, http://dx.doi.org/10.1016/j.swevo.2018.04.011.
[40] X. Li, S. Ma, Y. Wang, Multi-population based ensemble mutation method for single objective bilevel optimization problem, IEEE Access 4 (2016) 7262–7274.
[41] G. Wu, R. Mallipeddi, P.N. Suganthan, R. Wang, H. Chen, Differential evolution with multi-population based ensemble of mutation strategies, Inform. Sci. 329 (2016) 329–345.
[42] E. Alba, Parallel evolutionary algorithms can achieve super-linear performance, Inform. Process. Lett. 82 (1) (2002) 7–13, http://dx.doi.org/10.1016/S0020-0190(01)00281-2, evolutionary Computation.
[43] S.K. Nseef, S. Abdullah, A. Turky, G. Kendall, An adaptive multi-population artificial bee colony algorithm for dynamic optimisation problems, Knowl.-Based Syst. 104 (2016) 14–23.
[44] R. Tanabe, A. Fukunaga, Evaluation of a randomized parameter setting strategy for island-model evolutionary algorithms, in: Evolutionary Computation (CEC), 2013 IEEE Congress on, IEEE, 2013, pp. 1263–1270.
[45] A. Godio, Multi population genetic algorithm to estimate snow properties from GPR data, J. Appl. Geophys. 131 (2016) 133–144.
[46] S. Biswas, S. Das, S. Debchoudhury, S. Kundu, Co-evolving bee colonies by forager migration: A multi-swarm based artificial bee colony algorithm for global search space, Appl. Math. Comput. 232 (2014) 216–234.
[47] N. Kratzke, P.-C. Quint, Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study, J. Syst. Softw. 126 (2017) 1–16.
[48] P. Salza, F. Ferrucci, An approach for parallel genetic algorithms in the cloud using software containers, arXiv preprint arXiv:1606.06961.
[49] P. Salza, F. Ferrucci, F. Sarro, Develop, deploy and execute parallel genetic algorithms in the cloud, in: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, 2016, pp. 121–122.
[50] A. De Lucia, P. Salza, Parallel Genetic Algorithms in the Cloud (Ph.D. thesis), University of Salerno, 2017, URL http://elea.unisa.it/h{and}le/10556/2577.
[51] P. Dziurzanski, S. Zhao, M. Przewozniczek, M. Komarnicki, L.S. Indrusiak, Scalable distributed evolutionary algorithm orchestration using docker containers, J. Comput. Sci. (2020) 101069.
[52] H. Khalloof, W. Jakob, J. Liu, E. Braun, S. Shahoud, C. Duepmeier, V. Hagenmeyer, A generic distributed microservices and container based framework for metaheuristic optimization, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2018, pp. 1363–1370.
[53] J.J. Merelo, C.M. Fernandes, A.M. Mora, A.I. Esparcia, Sofea: a pool-based framework for evolutionary algorithms using couchdb, in: Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, ACM, 2012, pp. 109–116.
[54] J.L.J. Laredo, A.E. Eiben, M. van Steen, P.A. Castillo, A.M. Mora, J.J. Merelo, P2P evolutionary algorithms: A suitable approach for tackling large instances in hard optimization problems, in: 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008. Proceedings, 2008, pp. 622–631, http://dx.doi.org/10.1007/s00500-008-0297-9, URL http://www.springerlink.com/content/3j84234147m06241.
[55] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education India, 1995.
[56] N. Hansen, A. Auger, O. Mersmann, T. Tusar, D. Brockhoff, COCO: a platform for comparing continuous optimizers in a black-box setting, 2016, arXiv preprint arXiv:1603.08785, URL https://arxiv.org/abs/1603.08785.
[57] S. Zhao, P. Dziurzanski, M. Przewozniczek, M. Komarnicki, L.S. Indrusiak, Cloud-based dynamic distributed optimisation of integrated process planning and scheduling in smart factories, in: Proceedings of the Genetic and Evolutionary Computation Conference, 2019, pp. 1381–1389.
[58] F.-A. Fortin, F.-M.D. Rainville, M.-A. Gardner, M. Parizeau, C. Gagné, Deap: Evolutionary algorithms made easy, J. Mach. Learn. Res. 13 (Jul) (2012) 2171–2175.
[59] H. Faris, I. Aljarah, S. Mirjalili, P.A. Castillo, J.J. Merelo, EvoloPy: An open-source nature-inspired optimization framework in python, in: Proceedings of the 8th International Joint Conference on Computational Intelligence - Volume 1: ECTA, (IJCCI 2016), 2016, pp. 171–177, http://dx.doi.org/10.5220/0006048201710177.
[60] M. García-Valdez, J.J. Merelo, Benchmarking a pool-based execution with ga and pso workers on the bbob noiseless testbed, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2017, pp. 1750–1755.
[61] M. García-Valdez, L. Trujillo, J.J. Merelo-Guérvos, F. Fernández-de Vega, Randomized parameter settings for heterogeneous workers in a pool-based evolutionary algorithm, in: International Conference on Parallel Problem Solving from Nature, Springer, 2014, pp. 702–710.
[62] M. El-Abd, M.S. Kamel, Black-box optimization benchmarking for noiseless function testbed using particle swarm optimization, in: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, 2009, pp. 2269–2274.
[63] M. Nicolau, Application of a simple binary genetic algorithm to a noiseless testbed benchmark, in: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, 2009, pp. 2473–2478.
[64] L. Araujo, J.J. Merelo Guervós, C. Cotta, F.F. de Vega, Multikulti algorithm: Migrating the most different genotypes in an island model, arXiv preprint arXiv:0806.2843.



Dr. García-Valdez is a full-time research professor at the Tijuana Institute of Technology. He is interested in the personalization of interactive systems, voluntary computing, parallel evolutionary computation, interactive evolutionary computation.

JJ Merelo is professor at the university of Granada, where he obtained a degree in Theoretical Physics and a Ph.D. in Physics in 1994. He is mainly interested in evolutionary algorithms, open source software and complex systems.