EvoSwarm: An Event-based Architecture for 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*

Abstract

Splitting a population into multiple instances is a technique that has been used extensively in recent years in order to help improve the performance of nature-inspired optimization algorithms. Work on several populations can be done in parallel, and also asynchronously, a fact that can be leveraged to create scalable implementations based on, among other methods, distributed, multi-threaded, parallel, and cloud-based computing. However, the design of these cloud-native, distributed, multi-population algorithms is not a trivial task. Coming from monolithic (single-instance) solutions, adaptations at several levels, from the algorithmic to the functional, must be made to leverage the scalability, elasticity, 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 to create from scratch a cloud-native optimization framework that is able to include multiple algorithms and use few parameters to work. This solution goes beyond current state of the art via a framework that is able to support different algorithms at the same time, work asynchronously, and also be easily deployable to any cloud platform. We evaluate the performance and scalability of this solution, together with the effect other design parameters had on it; in particular, the number and the size of pop-

[*]Corresponding author
Email addresses: `mario@tectijuana.edu.mx` (Mario García Valdez), `jmerelo@geneura.ugr.es` (Juan J. Merelo Guervós)

ulations with respect to problem size. The architecture and 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 kinds of algorithms, and setting a new baseline for scaling and performance in the cloud.

Keywords: Multi-population, Nature-inspired algorithm, Parallel Genetic Algorithms, Cloud-Computing, Event-driven architecture

1. Introduction

In the last decades, nature-inspired optimization algorithms have been successfully applied to solve complex real-world problems [1]. Algorithms inspired by natural processes include evolutionary algorithms (EAs) [2] and swarm intelligence (SI) [3], among others. These population-based algorithms share the common characteristic of using an initial set of random candidate solutions that are later used to generate a new set of candidates, using a nature-inspired heuristic. Popular EAs are Genetic Algorithms (GAs) [4, 5], Genetic Programming (GP) [2], Grey 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].

Population-based algorithms are intrinsically parallel, and can be implemented to run asynchronously. The fitness of each individual can be evaluated independently of others, and similarly, each population could evolve in isolation. Since earlier works, researchers have been proposing some form of parallelization [10] to increase the scalability of these algorithms. The island model was one of the first techniques proposed for parallelization, which lead to an increased performance [11, 12]. The concept was to divide a large population into communicated subpopulations. Since then, other population-based algorithms have adopted the concept, and researchers have found additional benefits besides the execution speed; these include avoiding a premature convergence and maintaining the diversity of the global population [13]. Researchers use the term multi-population based methods when generally referring to techniques using many populations as part of their optimization strategy. Moreover, parallel implementations can run in two ways: synchronous and asynchronous. When operations run in parallel but must maintain synchrony with the other operations, all operations need to finish to continue to the next step. For instance, in a parent/worker model, the parent needs to wait for all workers to finish their operations before moving to the next iteration. In contrast, in an asynchronous execution, operations are not synchronized with each other. Following the previous example, now the parent operation can continue to the next iteration, even if a worker has not finished its work. In the literature, 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 the parallel execution multi-population algorithms go from earlier hardware-based implementations using transputers [11], multi-threaded [20], and multi-core systems [21, 22]. And recently the focus has been on exploiting a higher number of processing units by using GPUs [23, 24], or distributed systems; techniques used to achieve this include web-based [17], map-reduce [25], grid [26, 27], voluntary computing [28], and cloud computing [29, 18, 30, 31]. 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 cost of administration, 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 in the cloud old-style monolithic applications, that is, applications built on a single stack of services that communicate by layers, to microservices [32] and serverless architectures [33, 34] that favor the parallel and asynchronous communication of heterogeneous resources; in the process, a new process for designing *cloud native applications* has been created, with brand new methodologies and techniques [35]. In these architectures, services or even functions are seen as single processing points, departing from the monolithic and even distributed paradigms, become a loosely coupled collection of *stateless functions* [36], that react to events and only come into existence while they are doing some processing. Systems developed with the patterns outlined above have the properties of a reactive system [37] and they are commonly more flexible, loosely-coupled, and scalable.

When designing efficient multi-population algorithms, researchers need to consider additional issues [38]. These include the number and size of populations, the interaction between them, the search area of each population, and the search strategy and parametrization of each population. We must take into account these high-level issues when designing a parallel architecture. Researchers and algorithm designers willing to adapt to the cloud their multi-population solutions face the 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.
- To scale the system, when it is possible, they must use stateless functions for processing.
- They need to establish a workflow of local development and prototyping while having the option of deployment in a cloud provider. It is desirable to use modern development 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.

Taking these factors into account, in this work, we present EvoSwarm, a multi-container Docker application that reactively processes isolated and heterogeneous populations. The platform uses the Docker Compose tool for defining and running experiments as multi-container applications. EvoSwarm is also released with a free licence in `https://github.com/mariosky/EvoSwarm`. We preferred the use of a container-based application for this prototype because it is more suitable for development and replication without the need for additional components. Nevertheless, the application is compatible with other orchestration and deployment technologies if more scalability is needed. With additional configuration settings, applications can be deployed to a Docker Swarm or a Kubernetes system.

Docker Container images host the stateless functions responsible for running a search strategy on several populations, these functions could also be ported directly to a Function as a Service (FaaS) [39] implementation. Researchers then define the configuration of services used by a multi-population algorithm using a YAML file. Any researcher can deploy and start all the services from this configuration file with a single command. Once the services are up, multiple instances of an experiment can be sent to a queue for their execution. The platform includes containerized services that are compatible with other container orchestration technologies like Kubernetes and Docker Swarm. The services included are a Redis Server used for the Message Queue implementation and logging and an Experiment Controller implemented in Python. Developers of population processing functions have the freedom to implement new population-based algorithms or operators in the language they choose inside a container, as they only need to subscribe to a channel of the Message Queue and have the ability to read the population and parameters as JSON file.

This paper extends our earlier publication on the topic [19], and we highlight the main contributions as follows:

- 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 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 the performance of their algorithms with other works.
- Lastly, we present an empirical study to validate the applicability of our application, measuring the execution time, speedup.

The organization of the paper is 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 . Section 6 describes the design of the empirical evaluation we designed to assess the effectiveness of the method, and in Section 7, we report and discuss the results. Finally, we offer the conclusions of this paper and suggestions on 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 [38].This relative isolation helps in maintaining 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].

However, in most cases, algorithms applied to each population are homogeneous, or at any rate, the same variant of the algorithm. As long as this parallel operation is not synchronous, other population-based algorithms, or, as a matter of fact, any algorithm, 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 problem of finding the correct parameter settings for an algorithm; 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 the individual parameters of each population is compensated by those selected in other populations [40, 44].

In this work, we will use random settings within a specific range as results have shown this is a valid solution to this problem [45].

Some parameters, specially the population size, are kept fixed in order to control more easily the execution of the algorithm. For instance, by having the size of populations fixed, it is easier to control the number of evaluations and the communication costs, when the algorithm is in operation.

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, 46, 47]. 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 on first sight containers look like simply a method that allows easy shipping of applications, 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. On the other hand, containerization also allows, through the use of orchestration engines like Docker Swarm or Kubernetes, or through simple, non-self-scaling specifications using Docker Compose, a definition of all the relationships between different parts of the application, or services. The full realization of Docker-based architectures has led to the creation of a series of patterns that, together, conform cloud-native application development [48].

However, in the field of evolutionary algorithms, this has been a slow realization, from the origins where Salza and Ferrucci [49, 50] proposed an architecture that carried evolutionary algorithms to the cloud, developed further in [51] which starts 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. The way these containers are organized, however, is rather classical in nature: a master-worker approach, communicated using RabbitMQ, where replicated workers perform the tasks in parallel. They called their approach AMQPGA, inspired by the protocol, AMQP, used by RabbitMQ, the messaging system. Other EAs topologies have also been used in the cloud: [52] adapts an island model to a container architecture, in a system that is functionally equivalent to classical models, but with the added advantage of being easy to deploy and run.

Later approaches use microservices [53], or were systematized [18] achieving speed-up through the creation of an abstraction layer over the evolutionary algorithm services, and also automation of the whole workflow. This research bumped into some problems, since speedup is limited, mainly due to the non-automatic scaling of services and also the presence of a single master which is the one that runs the evolutionary algorithms. However, this master-worker architecture can be turned around with other kind of evolutionary architectures, such as pool-based systems [30, 54, 16] where a better match to the inherently heterogeneous nature of cloud-based architectures. Pool-based systems are closer to serverless systems [36], and they pull from the pool and run whole algorithms, instead of just evaluation. The pool from where they pull is the common store of results, and can be accessed in an asynchronous way, which suits better the nature of cloud systems.

4. Proposed Method

In this section, we present the design and a container-based implementation of the model we propose for the execution of 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. As a first step, we describe the general architecture first, and then every component in detail.

The most elemental 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 is going to be triggered when a message is pushed to it. The two components at each end of the queue 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 of the reasons for this is the use of functions that have no secondary effects or access global storage, 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, and thus, there is no difference between having one or many copies of the same function pulling work from the queue all at the same time, because no side effects, including unwanted resource locking problems, are going to result from this concurrency. Based on these general principles we define next the proposed model.

4.1. Event Driven Model

Based on a reactive architecture we proposed in a previous work [19], we now describe the general architecture shown in Figure 1. Again, we can explain the model using the analogy of producers and consumers of messages. First, we can see that there are 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. Also, the architecture has at least four processes indicated in the diagram as swimlanes: First, the `Setup` process, responsible for the reading a configuration file, and creating the initial populations. Second, the `Controller` process, responsible for the migration between populations, and keeping track of the iterations of the algorithm. 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

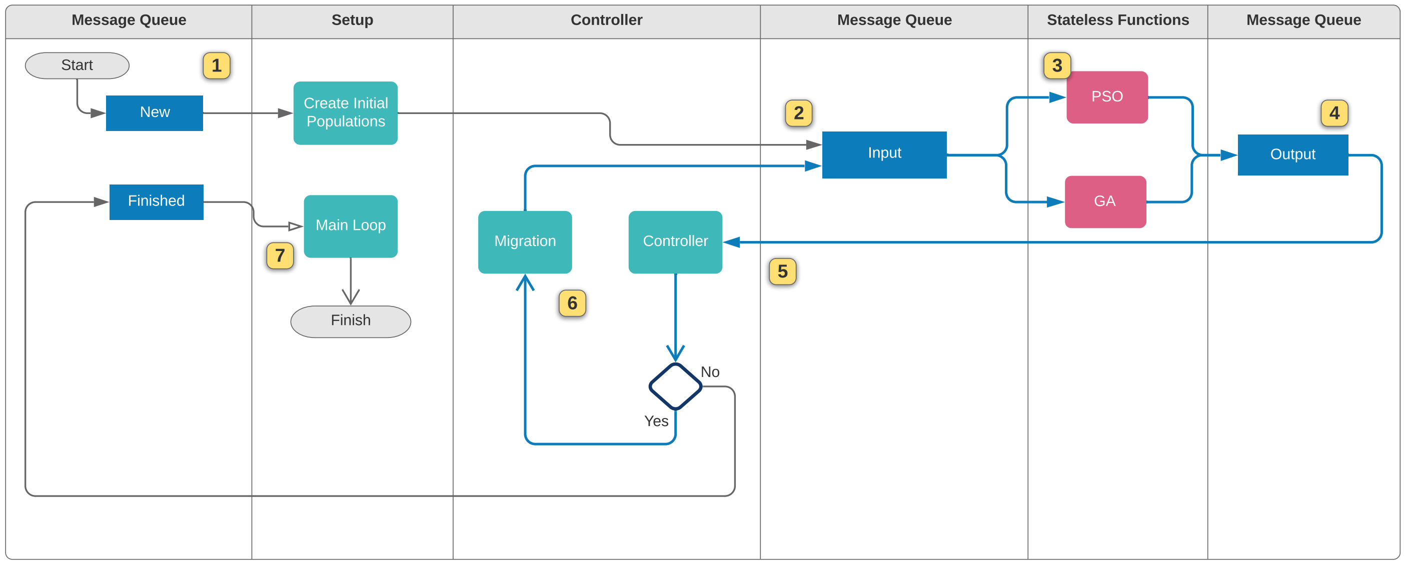


Figure 1: The proposed general architecture, showing each process in a swimlane, message-dataflow, message queues, and high-level tasks for each process.

from the `Experiments` queue, this is not shown in the diagram. The configuration message includes all the parameters needed to execute the algorithm, among other things, 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 5.4. 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 they can be consumed by Stateless functions responsible for the execution of the search.
3. One or more `Stateless functions` are constantly pulling population messages from the `Input` queue. Taking the population data and metadata as parameters, these functions are responsible for the actual execution of the optimization algorithm. 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 packed again along with additional metadata about the execution of the algorithm. 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 and if an optimal solution has been found or the maximum number of iteration has been reached it signals the stop of the execution.
6. Otherwise, it passes the stream of messages to a migration process, where populations are mixed with each other. New populations are generated from this migration, and they are again pushed to the Input queue to

continue in a loop. The `Controller` is also responsible of logging the metadata received with the messages.

This reactive architecture has the following advantages:

- An important aspect of 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 a different search strategy, or using its own 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 basic 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 a multi-threaded execution.

On the other hand, there are several caveats as well: It is more costly to move entire populations as messages than only passing certain 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 size of populations. Pool-based algorithms also suffer from this drawback, but web based implementations have been working with continuous optimization problems of 1000 dimensions. But, in other cases this is not a viable approach.

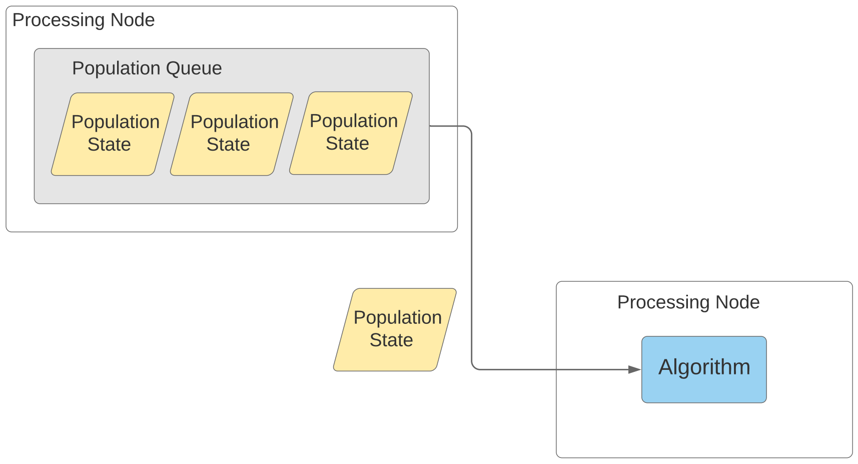


Figure 2: Population State and dataflow between processing nodes of a message-based algorithm.

4.2. Comparison with other cloud-based works

Next, we compare the message-driven architecture we presented against four other proposals for multi-population algorithms found in the literature. We center the comparison on the coupling and communication between the main components: populations, processing nodes, and algorithms. In Figure 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 Figure 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 form of execution, 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

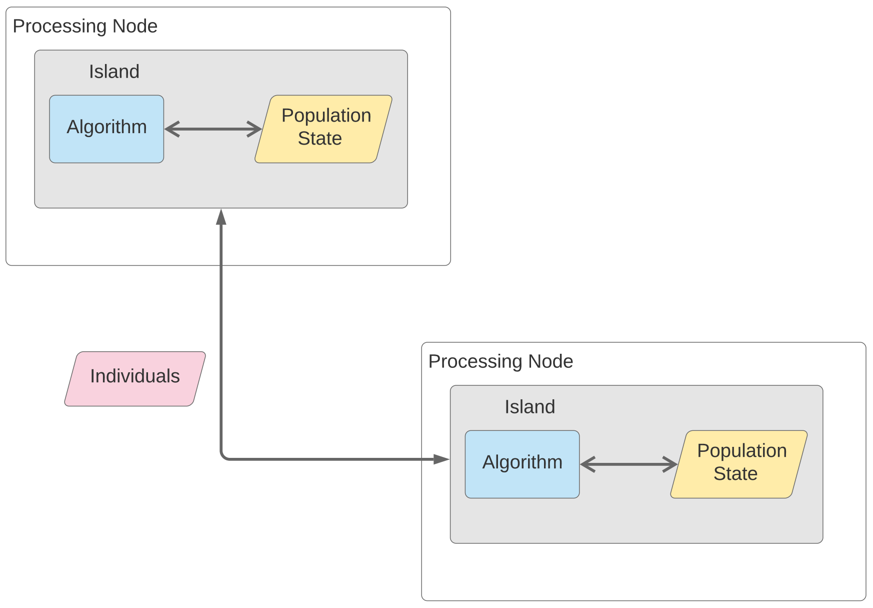


Figure 3: Population State and dataflow between processing nodes of a classic Island-based multi-population algorithm.

simply be aware of these new nodes. This is solved in part in Peer to Peer (P2P) evolutionary algorithms [55], but 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 the use of a central repository or pool of individuals that is available to all processing nodes. In Figure 4 we show the main components of a pool-based multi-population algorithm. Although algorithms and population state remains 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 pro-

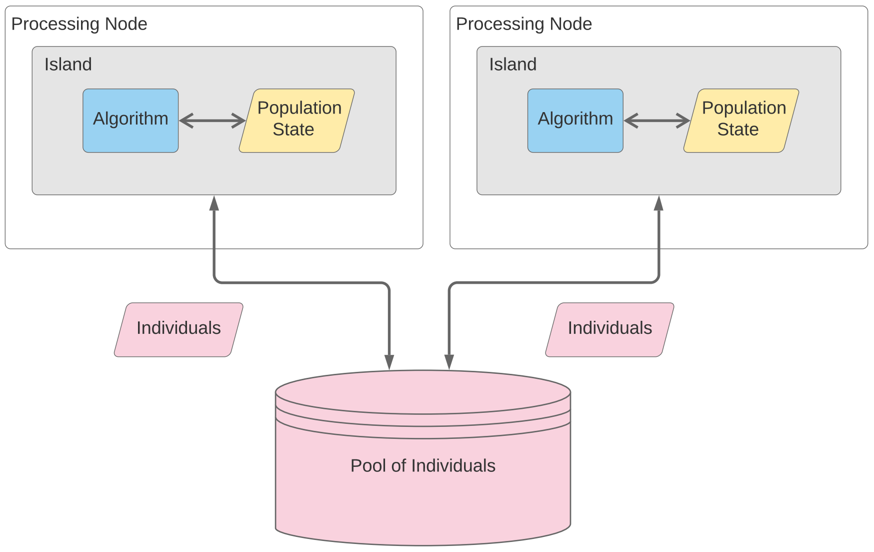


Figure 4: Population state and flow of data between processing nodes of a Pool-based multi-population algorithm.

cessing nodes, the system needs additional communication and processing for keeping track of each population.

We show another pool-based approach in Figure 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 state of the population is lost. But having the algorithm decoupled opens the possibility of implementing the system using serverless functions.

5. Container-based application

We have mentioned several advantages of a cloud-native application for running multi-population based algorithms, but there are in our experience several

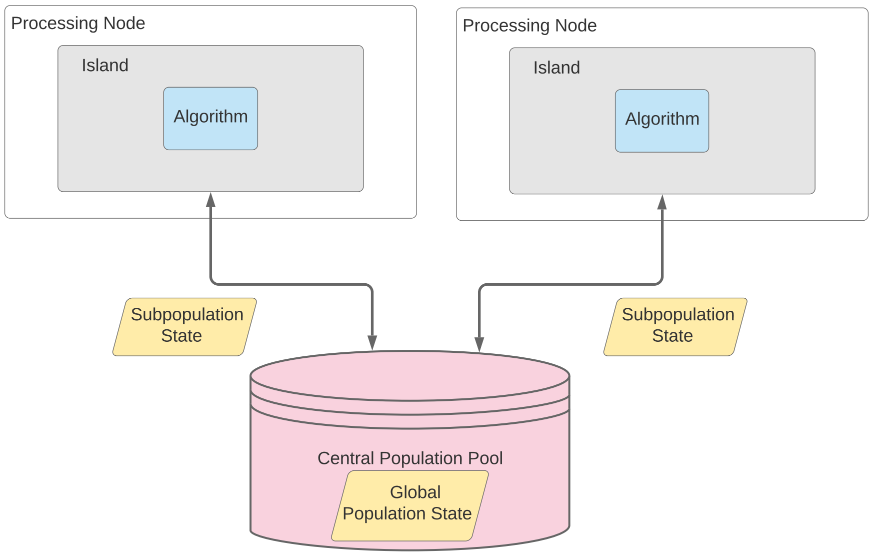


Figure 5: Population State and dataflow between processing nodes of the EvoSpace algorithm.

practical issues of this approach. In general, cloud platforms have free-tiers for their services, and many offer academic discounts to students. But once we pass to the pay-as-you-go tier, computing resources need to be managed, and, where possible, its cost minimized. First, in many cases, a credit-card is needed to guarantee the payment of the resources consumed and someone responsible for assigning quotas and, more importantly, special keys for using the cloud's API. If these keys are compromised or shared by their holders, additional costs or administrative procedures are in order. For those cases when a prototype platform is needed, for instance, students developing a new algorithm variant or experimenting with a published work, a container-based platform can be more practical. Container technology enables researchers the local deployment of all the infrastructure needed to run the experiments. Later, if needed, the same system could be deployed to the cloud.

The main advantage of container deployments over virtual machine instances is that they add much less overhead to communication between them, and they are much more economical to deploy. Scaling and monitoring can be done much more easily through cloud orchestration systems such as Kubernetes; these are the reasons why we chose it for this work.

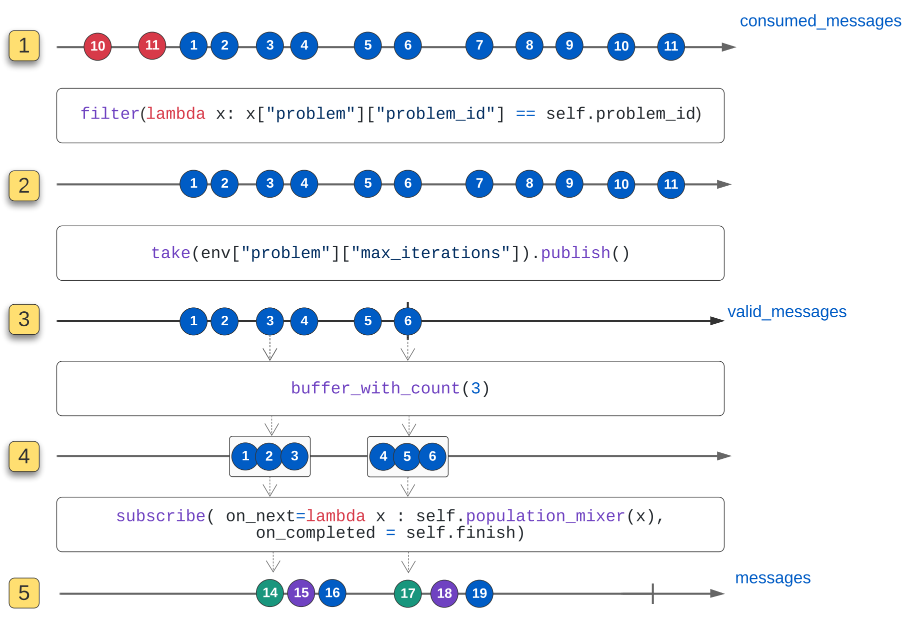
In this section, we propose a design of a reactive container-based application for executing multi-population based optimization experiments. For the design, we followed the design patterns highlighted in the previous section, and again we can go back to Figure 1 and use it as a guide for explaining the main components and their interactions.

5.1. Message Queue Container

In this work, we implemented all message queues in the Redis memory store. Early cloud architectures [51] used queues for communication, although in that case RabbitMQ was the chosen tool. Redis is faster, and also can act as a data store from where we can control the state of the application. 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 operation 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 Figure 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 `https://github.com/ReactiveX/RxPY` an it is based on the Observer pattern [56] and functional programming.



```
filter(lambda x: x["problem"]["problem_id"] == self.problem_id)
```

```
take(env["problem"]["max_iterations"]).publish()
```

```
buffer_with_count(3)
```

```
subscribe( on_next=lambda x : self.population_mixer(x),
    on_completed = self.finish)
```

Figure 6: Marble diagram for the Reactive Python implementation of the Controller

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 Figure 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. Normally, a transformation results in another `Observable`, again emitting new results. For asynchronous programming, this approach is more elegant than nested callbacks, that are more difficult to code and debug.

Now we proceed to explain the reactive implementation of the controller. In Figure 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`. In this particular period, we can see that the stream receives two populations from another experiment, and 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 6 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 to 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 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.

Algorithm 1 Migration

```
1: procedure CXBESTFROMEACH($pop_1, pop_2$)
2:     $pop_1.sort()$
3:     $pop_2.sort()$
4:     $size \leftarrow min(len(pop_1), len(pop_2))$
5:     $cxpoint \leftarrow (size - 1)/2$
6:     $pop_1[cxpoint:] \leftarrow pop_2[: cxpoint + 2]$
7:     return $pop_1$
8: end procedure
```

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

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). Once the function returns a new population, the worker pushes the results to the `Output` queue, finishes, and passes the execution to the main loop. This process is shown in Figure 7.

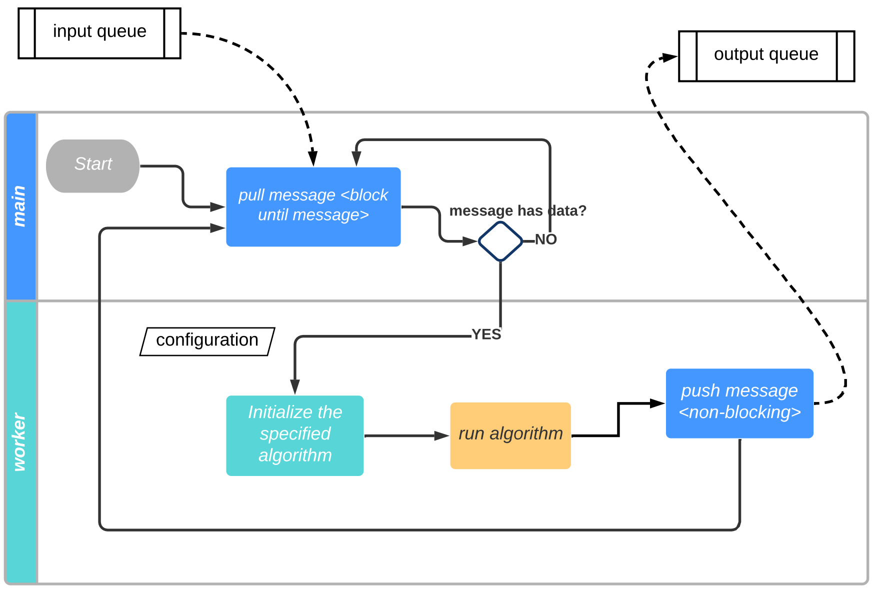

input queue
output queue
main
worker
Start
pull message <block until message>
message has data?
NO
YES
configuration
Initialize the specified algorithm
run algorithm
push message <non-blocking>

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

5.4. Experimental Workflow in EvoSwarm

Figure 8 shows the workflow users can follow to run experiments in EvoSwarm. The first step is to develop the search strategy algorithms that are going to be used by the multi-population algorithm. Algorithm developers can upload the container definition or images directly to DockerHub; another option is to upload the container image definition (Dockerfile) to GitHub and define a trigger to re-upload the image to DockerHub; this can be defined in DockerHub (or any other registry) too. Docker Compose configuration files can reference these containers. When a user needs to deploy the application, he or she needs to edit the `docker-compose.yml` file to specify the type and number of containers needed for the application. The `docker-compose` application is responsible for

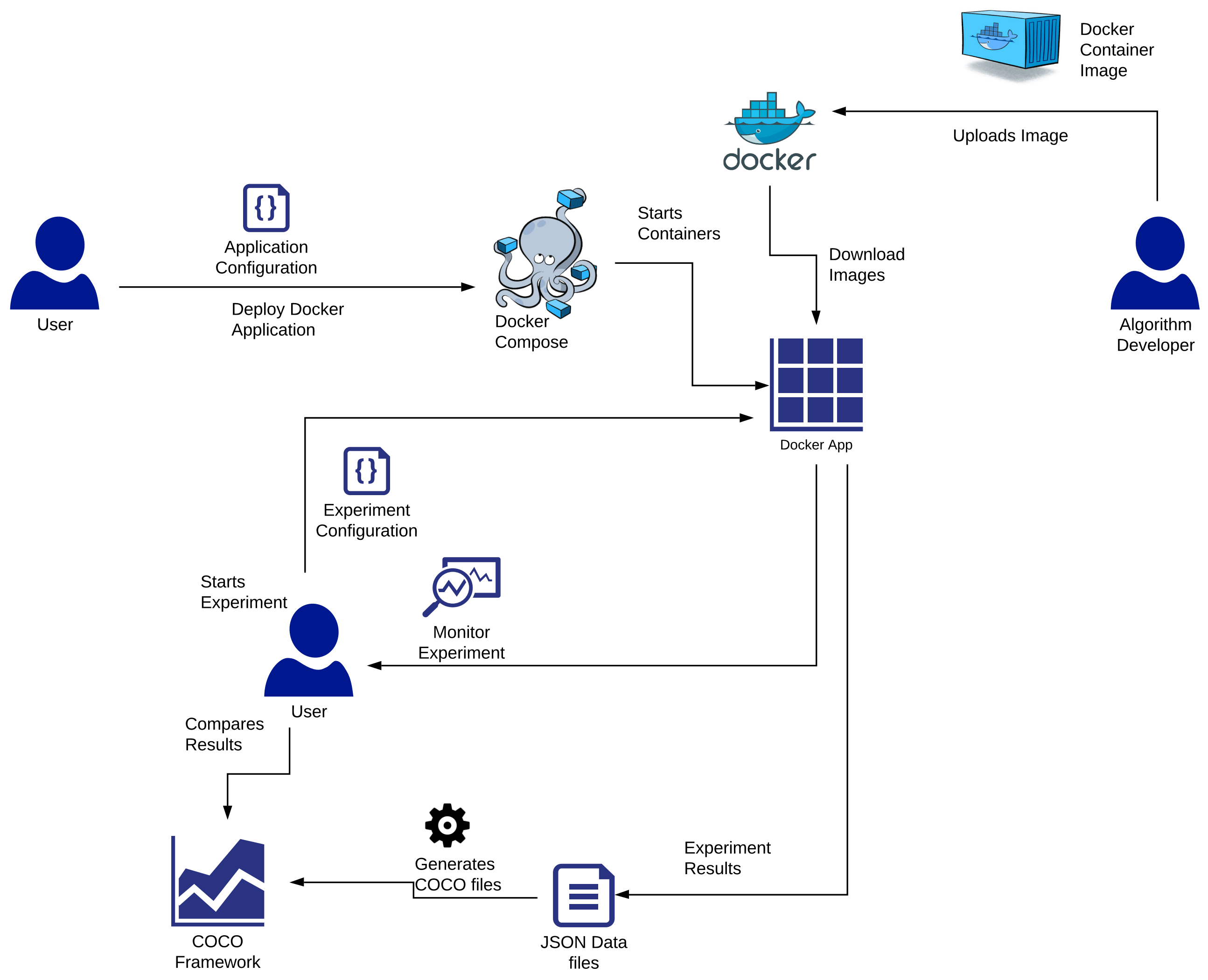


Figure 8: EvoSwarm Workflow

starting all the containers, downloading new versions if it is necessary. After starting the application, users can push Experiment Definition files and start monitoring the execution in the terminal. After the 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 in a JSON format. Researchers can use these files to analyze and plot experiment data. Python scripts are included in the repository to plot the running times for the experiments, and all other plots used in this paper; there is also a script to generate the files needed by the COCO framework [57], which can generate standard comparisons against other methods.

6. Experimental Setup

We aimed to learn if the proposed solution can efficiently improve the scalability and performance of population-based optimization algorithms. Hence, in this section, we verify the following:

- Whether we can improve the execution time of the algorithm by adding populations and serverless functions to the system, in particular, what is the effect of these changes on the scalability of the system?
- Having a multi-population enabled platform, with heterogeneous populations and the support for mixing search strategies, increases the performance of the search by needing fewer function evaluations than a homogeneous setting?

To validate these questions, we used benchmark functions from the Continuous Noiseless BBOB testbed, which is part of the Comparing Continuous Optimizers (COCO) framework [57]. 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 of the testbed utilizes 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 the performance of algorithms, generating data sets, tables, and reports for an experiment. There is a repository[1] 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 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). For the GA implementation, we used the DEAP library [58] and for the PSO, the EvoloPy library [59]. Both Python libraries are open-source. In EvoSwarm these two algorithms are implemented as stateless functions.

Next, we show the parameters for each algorithm, with Table 1 for the GA and Table 2 for PSO. We obtained these parameters following the same method as in a previous work [60]. To obtain the parameters, we tested first on the Rastrigin separable function with five dimensions. After about fifteen experiments, the most challenging targets were achieved for this particular function. We tested again with functions one to three, and after obtaining favorable results, the PSO and GA algorithm parameters were set. In the GA, we randomly set the mutation and crossover probabilities to have more heterogeneous workers; in these parameters, the range of values is specified. We did not change these parameters during the experiments, and only the population size and number of generations were provided as parameters.

The EvoSwarm application uses configuration files to run the noiseless BBOB

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

Table 1: 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 2: EvoloPy PSO Parameters

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

testbed experiments. As a first step, we must deploy the docker application using a `docker-compose.yml` file, where the number of worker containers is specified. Then a JSON file containing the configuration parameters of the experiment has to be provided. Table 3 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 all populations will run the GA algorithm. Next, we specify a list indicating which of the 24 benchmark functions will be tested. In the example, the experiment will use the first five functions. The *Instances* parameter indicates how many instances of each function will be tested. *Instances* has a default value of 15. We used this value because it is the standard for the BBOB benchmark [57]. 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 function evaluations (FEs) we mentioned earlier, we define for each dimension: the number of popu-

Table 3: Experiment configuration example

Parameter			Type	Example
Worker Containers (Docker Compose)			`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
40	80	250	5	40

lations 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 will be performed. All these parameters give us the maximum number of FEs that will be performed. For instance, for $D = 2$, the maximum number of FEs is $200,000$, which is the same as $40 * 50 * 10 * 10$.

As we mentioned before, 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., 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 effectively scale on multiple nodes, tending to linear scalability. 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.

Table 4: Parameters used in the experiments with ten populations

Dimension	2	3	5	10	20	40
Generations	40	25	28	50	66	80
Population Size	50	60	60	70	100	125
Populations	10	10	10	10	10	10
Iterations	10	20	30	30	30	40

Table 5: Parameters used in the experiments with five populations

Dimension	2	3	5	10	20	40
Generations	40	25	28	50	66	80
Population Size	100	120	120	140	200	250
Populations	5	5	5	5	5	5
Iterations	10	20	30	30	30	40

- The size of populations and the number of generations: These parameters naturally increase the execution time.
- The complexity of the algorithm: For instance, in our case, the PSO implementation has a lower execution time than the GA.

To evaluate the effect of these parameters on the scalability of the system, we propose an experiment in which we selected f_4 (Skew Rastrigin-Bueche separable) from the BBOB testbed. This function has been used because it is computationally demanding, and in higher dimensions has been difficult for PSO [61] and GAs [62] to solve with the required FEs. As we are only comparing in terms of execution time, the results from a single function can be better understood, since there are fewer factors involved.

We executed two sets of experiments with five and ten initial populations, repeating each experiment using 1, 2, 4, and 8 workers. We kept the same maximum number of FEs, changing the relevant parameters for this. See Tables 4 and 5 for the complete list.

7. Experimental Results

We present the results of the experiments in this section. with results summarized in Figure 9. The experiment consisted of executing the same benchmark on EvoSwarm twice but changing the number of populations used in each run. This is mainly intended as a showcase of the flexibility of the tool presented in this paper, but we also wanted to test how the number of workers affected the time needed to reach a solution. Adding more populations might add more overhead, so this might have a noticeable effect. The left column in that figure has the results when using five populations, and the right side shows the results for ten. In principle the main issue is how to find the right number of workers for a particular problem, but in general we wanted to check the communication overhead when the size of the problem increases and how the presented framework reacted to it.

Charts show the execution time achieved by a certain number of workers, for every problem dimension, which, besides, changes the population size. Each box-plot indicates the time required for the execution of 15 function instances, in seconds. Each row has a different scale on the y-axis, to compensate for the dimension increments. It is important to notice that the execution time can also be affected by the algorithm finding a solution with an error smaller than the threshold, before completing all the FEs.

As expected, and in most cases, increasing the number of workers reduces the time required for the execution of an instance, mainly when using the same parameters (number of populations and number of workers), that is, performance scales with the number of workers for a fixed problem and number of sub-populations used. However, when we compare the two options, we find some subtle differences: When using one or two workers, using 5 populations takes less time in most cases. A reason for this could be that the migration of individuals takes more time to reach the worker. For instance, when a single worker has 10 populations, only after processing every one of them, it will start receiving mixed populations. Although, in all cases, and even a single worker,

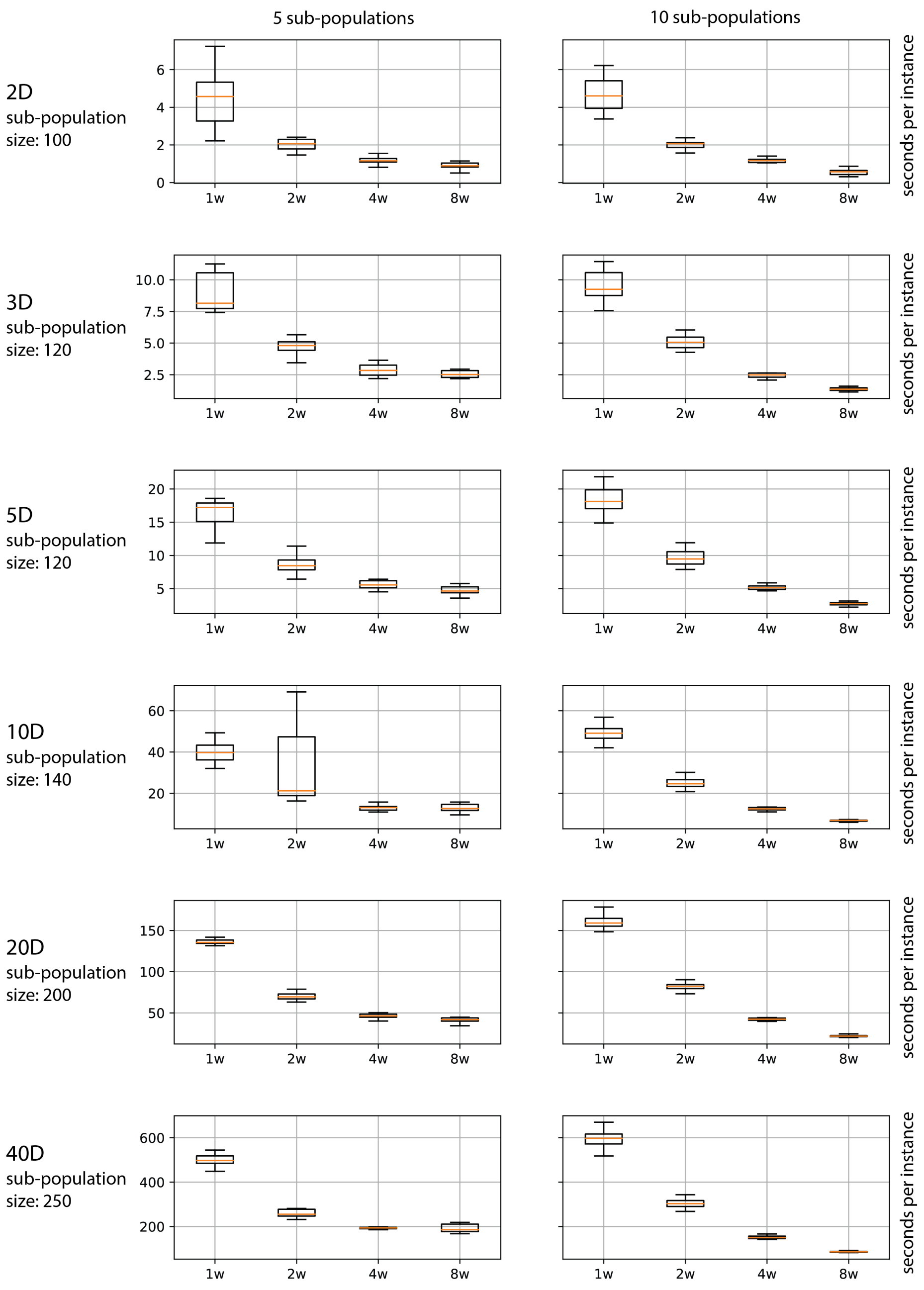


Figure 9: Time required, in seconds, to complete an instance of the function, with different number of workers and populations sizes

populations are heterogeneous, this advantage takes more time to propagate. With four workers, almost the same results can be achieved regardless of our selection between five or ten populations. Also, there is not much difference between 4 and 8 workers, when using five populations, which might indicate that some workers are idling while populations are being processed by other workers. In general, we can observe that keeping a ratio close to 1:1, but a bit better than that, between the number of workers and populations, better exploits the computing resources available. This also decreases the relation between the overhead needed to migrate and process incoming and outgoing population and the total computing time: more populations will need more time to do that.

Another additional issue is related to the algorithm itself. Many more workers than populations will have a wider range of states of evolution in the experiment; populations that will be mixed will have very different fitness states, which means that a part of them will probably have been not so useful, following the intermediate disturbance hypothesis. When the number of workers is increased, the difference between the state of evolution of populations will be intermediate, leading to better results; eventually, when there are as many workers as populations, the state will be quite similar, which is shown with a lower gap in the time won adding more workers.

8. Conclusions

In this paper we have presented the EvoSwarm framework that implements an event-based, cloud native architecture suitable for hosting any kind of multi-population optimization algorithm. The framework is based in industry-standard development tools such as Docker and Docker Compose, making easy to develop and deploy multi-algorithm distributed experiments.

The set up of this kind of algorithms will have a big influence on the results, since diversity is important for this kind of algorithms and its creation and destruction play a big part of the algorithm's performance. One of the main levers to control it in EvoSwarm is via the number of workers that will be allocated from the beginning, which is why we have performed an experiment mainly intended as a proof of concept, but also devoted to show how two parameters in the framework, number of workers and number of populations used to initially seed the system, interact with each other. The conclusion of this experiment is that there is an interesting interplay, with the most benefit being obtained with a number of populations equal to the number of workers or slightly inferior or superior, which brings algorithmic benefit (experiment taking less time because it needs less evaluations) and also performance benefits (experiment taking less time because evaluations are being split between different workers.

As a conclusion, EvoSwarm is a framework designed and tested according to industry standards which is able to bring many benefits to the bioinspired algorithm community, by showing the possibility of creating new, high performance, mixed algorithms, which can also be easily deployed to the cloud, or locally to systems that simply need to have Docker and Docker Compose installed.

It also opens new possibilities. From the design point of view, it would be interesting to try and use automatic scaling, instead of setting the number of workers by hand. This would need a certain amount of redesign, the most important of which would be using container orchestration systems such as Kubernetes or Docker Swarm. 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 algorithms would be a possibility, but also 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 find out which sets of algorithms are a better match to each other. All these will be explored as future lines of work.

9. Acknowledgements

This paper has been supported in part by projects TecNM-5654.19-P and DeepBio (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, Advances in engineering software 69 (2014) 46–61.

[7] D. Karaboğa, S. Ökdem, A simple and global optimization algorithm for engineering problems: differential evolution algorithm, Turkish Journal of Electrical Engineering & Computer Sciences 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 computing 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. Dissertation, University of Michigan.

[13] C. Li, T. T. Nguyen, M. Yang, S. Yang, S. Zeng, Multi-population methods in unconstrained continuous dynamic environments: The challenges, Information Sciences 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. doi: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, pp. 1372–1379. doi:10.1109/CEC.2008.4630973.

[18] P. Salza, F. Ferrucci, Speed up genetic algorithms in the cloud using software containers, Future Generation Computer Systems 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] 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.

[21] R. Serrano, J. Tapia, O. Montiel, R. Sepúlveda, P. Melin, High performance parallel programming of a GA using multi-core technology, Studies in Computational Intelligence 154 (2008) 307–314. doi:10.1007/978-3-540-70812-4_17.

[22] X. Lai, Y. Zhou, An adaptive parallel particle swarm optimization for numerical optimization problems, Neural Computing and Applications 31 (10) (2019) 6449–6467.

[23] Y. Tan, K. Ding, A survey on gpu-based implementation of swarm intelligence algorithms, IEEE transactions on cybernetics 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, International Journal of Innovative Computing, Information and 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, Applications of Evolutionary Computation (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 Generation Computer Systems 26 (4) (2010) 633–644.

[27] D. L. Gonzalez, F. F. de Vega, L. Trujillo, G. Olague, L. Araujo, P. A. Castillo, J. J. M. 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] M. García-Valdez, L. Trujillo, J.-J. Merelo, F. Fernández de Vega, G. Olague, The EvoSpace model for pool-based evolutionary algorithms, Journal of Grid Computing 13 (3) (2015) 329–349. `doi:10.1007/s10723-014-9319-2`.

[30] 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.

[31] D. Sherry, K. Veeramachaneni, J. McDermott, U. M. O'Reilly, Flex-GP: Genetic programming on the cloud, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 7248 LNCS (2012) 477–486. `doi:10.1007/978-3-642-29178-4_48`.

[32] J. Thönes, Microservices, IEEE Software 32 (1) (2015) 116–116. `doi:10.1109/MS.2015.11`.

[33] B. Varghese, R. Buyya, Next generation cloud computing: New trends and research directions, Future Generation Computer Systems 79 (2018) 849–861.

[34] B. Varghese, R. Buyya, Next generation cloud computing: New trends

and research directions, Future Generation Computer Systems 79 (2018) 849–861, cited By 2. doi:10.1016/j.future.2017.09.020.

[35] 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. doi:10.1145/2897073.2897713.

[36] 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 Generation Computer Systems.

[37] J. Bonér, D. Farley, R. Kuhn, M. Thompson, The reactive manifesto, Dosegljivo: http://www. reactivemanifesto. org/.[Dostopano: 21. 08. 2017].

[38] H. Ma, S. Shen, M. Yu, Z. Yang, M. Fei, H. Zhou, Multi-population techniques in nature inspired optimization algorithms : A comprehensive survey, Swarm and Evolutionary Computation 44 (July 2017) (2019) 365–387. doi:10.1016/j.swevo.2018.04.011.

[39] M. Roberts, Serverless Architectures (2016) 1–36.

[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, Information Sciences 329 (2016) 329–345.

[42] E. Alba, Parallel evolutionary algorithms can achieve super-linear performance, Information Processing Letters 82 (1) (2002) 7 – 13, evolutionary Computation. doi:https://doi.org/10.1016/S0020-0190(01)00281-2.

[43] S. K. Nseef, S. Abdullah, A. Turky, G. Kendall, An adaptive multi-population artificial bee colony algorithm for dynamic optimisation problems, Knowledge-based systems 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] 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.

[46] A. Godio, Multi population genetic algorithm to estimate snow properties from gpr data, Journal of Applied Geophysics 131 (2016) 133–144.

[47] 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, Applied Mathematics and Computation 232 (2014) 216–234.

[48] N. Kratzke, P.-C. Quint, Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study, Journal of Systems and Software 126 (2017) 1–16.

[49] P. Salza, F. Ferrucci, An approach for parallel genetic algorithms in the cloud using software containers, arXiv preprint arXiv:1606.06961.

[50] 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.

[51] A. De Lucia, P. Salza, Parallel genetic algorithms in the cloud.

[52] P. Dziurzanski, S. Zhao, M. Przewozniczek, M. Komarnicki, L. S. Indrusiak, Scalable distributed evolutionary algorithm orchestration using docker containers, Journal of Computational Science (2020) 101069.

[53] 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.

[54] 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.

[55] 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. `doi:10si.1007/978-3-540-85451-7`. URL `http://dx.doi.org/10.1007/978-3-540-85451-7_66`

[56] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.

[57] N. Hansen, A. Auger, O. Mersmann, T. Tusar, D. Brockhoff, COCO: a platform for comparing continuous optimizers in a black-box setting, arXiv preprint arXiv:1603.08785 (2016).
URL `https://arxiv.org/abs/1603.08785`

[58] F.-A. Fortin, F.-M. D. Rainville, M.-A. Gardner, M. Parizeau, C. Gagné, Deap: Evolutionary algorithms made easy, Journal of Machine Learning Research 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. `doi:10.5220/0006048201710177`.

[60] M. García-Valdez, 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. 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.

[62] 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.