J Grid Computing (2015) 13:329–349
DOI 10.1007/s10723-014-9319-2

The EvoSpace Model for Pool-Based Evolutionary Algorithms

Mario García-Valdez · Leonardo Trujillo · Juan-J Merelo · Francisco Fernández de Vega · Gustavo Olague

Received: 28 June 2014 / Accepted: 16 October 2014 / Published online: 7 November 2014

Abstract This work presents the EvoSpace model for the development of pool-based evolutionary algorithms (Pool-EA). Conceptually, the EvoSpace model is built around a central repository or population store, incorporating some of the principles of the tuple-space model and adding additional features to tackle some of the issues associated with Pool-EAs; such as, work redundancy, starvation of the population pool, unreliability of connected clients or workers, and a large parameter space. The model is intended as a platform to develop search algorithms that take an opportunistic approach to computing, allowing the exploitation of freely available services over the Internet or volunteer computing resources within a local network. A comprehensive analysis of the model at both the conceptual and implementation levels is provided, evaluating performance based on efficiency, optima found and speedup, while providing a comparison with a standard EA and an island-based model. The issues of lost connections and system parametrization are studied and validated experimentally with encouraging results, that suggest how EvoSpace can be used to develop and implement different Pool-EAs for search and optimization.

Keywords Pool-based evolutionary algorithms · Distributed evolutionary algorithms · Heterogeneous computing platforms for bioinspired algorithms · Parameter setting

M. García-Valdez
Instituto Tecnológico de Tijuana, Calzada Tecnológico S/N, Tijuana, BC, 22414, Mexico
e-mail: mario@tectijuana.edu.mx

L. Trujillo (✉)
Departamento de Ingeniería Eléctrica y Electrónica, Posgrado en Ciencias de la Ingeniería, Instituto Tecnológico de Tijuana, Calzada Tecnológico S/N, Tijuana, BC, 22414, Mexico
e-mail: leonardo.trujillo@tectijuana.edu.mx
URL: www.tree-lab.org

J.-J. Merelo
Departamento de Arquitectura y Tecnología de Computadores, Centro de Investigación en Tecnologías de la Información y las Comunicaciones, Universidad de Granada, Granada, Spain
e-mail: jmerelo@geneura.ugr.es
URL: http://citic.ugr.es

F. Fernández de Vega
Grupo de Evolución Artificial, Universidad de Extremadura, Extremadura, Spain
e-mail: fcofdez@unex.es

G. Olague
Centro de Investigación Científica y de Educación Superior de Ensenada, Ensenada, BC, Mexico
e-mail: olague@cicese.mx

1 Introduction

Grid Computing was developed by Foster and Kesselman in the early 1990s [17], an infrastructure designed to enable resource sharing from multiple locations, while also being coordinated to reach a common goal. Early Grid applications were initially motivated by the particular demands of advanced science and engineering applications [4], seeking a low cost alternative to dedicated data centers or clusters. What distinguished grid computing from conventional high performance computing systems, was that grids tend to be more loosely coupled, heterogeneous, and geographically dispersed [17]. With the ubiquity of the Internet, nowadays researchers have access to vast computational resources which are disperse and available using various technologies, including cloud computing [3, 44], peer-to-peer (P2P) networks [34], and web services [10].

On the other hand, Evolutionary Computation (EC) researchers require powerful computational resources, since EC algorithms tend to be computationally demanding. EC algorithms can benefit from grid computing infrastructures, by designing them as parallel, distributed and asynchronous systems. Several Evolutionary Algorithms (EAs) have been proposed that distribute the evolutionary process among heterogeneous devices, not only among controlled nodes within in-house clusters, but also in those outside the data center, in users' web browsers, smart phones or external cloud based virtual machines. This reach out approach allows researchers the use of low cost computational power that would not be available otherwise. However, such systems present several noteworthy challenges, such as the need to manage heterogeneous and unreliable computing resources, low bandwidth communications, as well as security and privacy issues.

This paper is part of an ever-growing effort in the field of EC to develop algorithms following this opportunistic approach to computing, allowing the exploitation of freely available services over the Internet or volunteer computing resources. To enable this arbitrary plug-in and plug-out of computational resources for distributed EAs, a new pool-based architecture has been proposed [19–21, 29–32, 43]. The basic idea behind Pool-based EAs (Pool-EAs) is that all the participants of the distributed algorithm share a common population stored in a central server, from which each participant can pull a set of tasks or data from the server, perform some work and then return the results back to the server. Pool-EAs differ from the closely related island model, mainly with regards to the responsibilities that are assigned to the server. In the island model the server is usually responsible for the interaction and synchronization of all the populations, while in a Pool-EA the server only receives stateless requests from isolated participants or clients. In this way, Pool-EAs are capable of an ad-hoc collaboration of computing resources. However, there are still several open issues that need to be addressed for the deployment of a Pool-EA system, particularly:

1. Work Redundancy: in a Pool-EA, two or more participants often take the same tasks or data to work on, possibly duplicating computational effort.
2. Starvation of the Population Pool: this happens when there are no more tasks or data in the server, participants must wait idly for tasks to become available.
3. Participants are Unreliable: participants might abandon their work, they might malfunction or loose their connection, thus delaying or failing to return their results to the server.
4. Parametrization: The number of parameters is incremented with respect to a traditional EA, with specialized parameters such as the number of tasks pulled by each participant, the size of the pool, the amount of time the server waits for a result, among others. Moreover, participants could also have internal parameters, that could be set homogeneously or heterogeneously.
5. Single Point of Failure: conceptually, there is a single central server managing the pool, a single point of failure.

Some of these issues have been widely studied in distributed systems research. Gelertner and Carriero [22] developed a model that could address issue (1), it is based on the concept of a tuple space, a repository of tuples or shared objects, that can be accessed concurrently by several processes. This tuple space works as a communication and coordination mechanism between distributed processes that interact with each other through these shared tuples. A distinctive characteristic of this model is that in order to change the state of a tuple, a process needs to atomically remove the tuple from the space and place it back only after it has released it; this inherently provides mutual exclusion since another process cannot access a tuple while it is being used. A benefit of such a model for Pool-EAs is that it forbids two clients from duplicating work. On the other hand, issues (2) and (4) are very specific to the Pool-EA approach. Finally, issues (3) and (5) are common for distributed platforms, and proper solution strategies will have to consider the particular requirements of the problem domain [2, 46]. However, issue (5) can be addressed at the technological level for Pool-EAs, without explicitly considering the specific evolutionary model; for instance, using hardware redundancy, distributed file systems or transparent data replication.

This paper presents the EvoSpace Model (ESM), a conceptual framework for the development of Pool-based EAs [21]. The ESM is built around a central repository or population store, incorporating some of the main principles of the tuple-space model. However, the ESM is not simply an implementation of the tuple-space model for EAs, since it incorporates additional mechanisms and concepts specifically tailored to address each of the above listed issues for Pool-EAs. In particular, issue (1) is solved by defining the interaction between distributed workers and the centralized pool using the tuple space principles. Issue (2) is solved by incorporating a reinsertion process, that determines when the central Pool will be starved, and takes proper corrective measures. Moreover, experimental work presented in this paper suggests that the ESM is not hampered by issues (3) and (4).

The ESM has two main components, a set of EvoWorkers and a single instance of an EvoStore. The EvoStore container manages a set of objects representing individuals in a EA population. EvoWorkers pull a subset of individuals from the EvoStore, making them unavailable to other workers. Moreover, individuals are removed from the EvoStore as a random subset or sample of the population, this differs from the tuple-space paradigm which removes only those tuples matching a pattern-based query. The ESM also includes a reinsert operation which allows a sample once removed from the population to be reinserted on-demand. Intuitively this can be seen as if individuals once removed from the EvoStore changed to a phantom state, and if needed they could be brought back to life. Once an EvoWorker has a sample to work on, it can perform a partial evolutionary process, and then return the newly evolved subpopulation to the EvoStore where the new individuals replace those found in the original sample; at this point replaced or reinserted individuals can be taken by other clients. Another possibility is that EvoWorkers pull a sample just to change the state of individuals and then return the sample, all of this without executing an EA loop. Example of this scenario can be found in interactive evolutionary algorithms, where users pull a sample of individuals from the EvoStore container to assign a fitness value [20, 43]. EvoWorkers perform their tasks in an asynchronous manner, contrasting with the traditional sequential and synchronous evolutionary model followed by most EAs. Indeed, new evolutionary models, such as the ESM, allow the design of Pool-EAs that include algorithmic features present in natural evolutionary systems, but that are difficult to reproduce in sequential and synchronous processes [13].

This paper presents a detailed description of the ESM, building upon our previous contributions [19, 21], and providing a comprehensive analysis of the system at both the conceptual and implementation levels. Furthermore, several design issues are addressed, that illustrate the robustness of the ESM. Particularly, the problems of lost connections and system parametrization are studied and validated experimentally, with encouraging results, that suggest how EvoSpace can be used to develop and implement different Pool-EAs for search and optimization.

The remainder of the paper proceeds as follows. First, a comprehensive discussion pertaining to related work is presented in Section 2. Afterwards, the ESM is detailed in Section 3 and a reference implementation is presented in Section 4. Three experimental validations are performed in Section 5 using genetic algorithm benchmarks: first, the ESM is compared with another distributed EA to illustrate some of its strengths and weaknesses; second, the ESM is compared with a standard EA based on its ability to find local optima and search efficiency; and finally, a cloud based deployment is used to evaluate the system's communication costs and scalability. Section 6 deals with two general issues associated with Pool-EAs in general, and ESM in particular, and proposes useful solutions which are validated experimentally. Finally, a summary of this work, conclusions and future work are presented in Section 7.

2 Related Work

In terms of parallelizing EAs, a large body of work has developed a variety of parallel models, with important practical and theoretical results [1]. However, our work focuses on the goal of distributing an EA over a grid, a much less explored topic in EA research. In particular, this section will contextualize Pool-EAs within the more general research area of distributed EAs, to correctly frame the contributions made in the proposed model.

Probably the most unobtrusive way to distribute an EA over an heterogeneous collection of computing resources is to run an EA over the Internet using web browsers. For instance, Klein and Spector [24] and Merelo et al. [33] developed EAs that distribute fitness function evaluations over the web using JavaScript on the client machine, such an approach comes with the benefit that connected clients do not need to install any special software on their systems, since the code is directly executed on the browser. Another recent example is given by Cotillon et al. [9]; they take the idea of browser-based EAs to the Android OS, which can be executed on a large number of mobile devices. Other common examples of web-based EAs come from the area of interactive evolution, where users visit a web page, browse candidate individuals and participate in the evolutionary process by evaluating individuals in artistic design problems [20, 26, 38, 43]. However, in these works the EA process is not necessarily executed on the browsers themselves, they are mostly used as practical interfaces to obtain user input, while the actual search is performed on the server.

Other researchers have exploited other network-based technologies to distribute an EA over a set of computing nodes. For instance, Cole et al. [8] uses the popular Berkeley Open Infrastructure for Network Computing (BOINC) to distribute an EA, using the volunteer computing model, where connected clients share idle CPU cycles with a research project. Another example is the work of Fernández-de-Vega et al. [15], who also distributes multiple EA runs using a volunteer computing network through BOINC. Garcia-Arenas et al. [18] used the popular file sharing service Dropbox as a storage server for an EA. Jiménez-Laredo et al. [18] uses a P2P architecture to distribute a Genetic Programming (GP) algorithm over a network. Similarly, the DREAM framework presented in [35] also distributes an EA using a P2P network based on mobile multi-agent systems. Another example is the ParadisEO object-oriented framework for the reusable design of parallel and distributed metaheuristics, including EAs, developed by Cahon et al. [6] using a distributed model based on the Farmer/Worker paradigm.

Recently, researchers have also begun to fully integrate EAs into the Cloud computing model. For instance, the Eureqa [1] GP-based modeling tool, originally developed by Schmidt and Lipson [37], and now marketed by Nutonian, easily integrates with Amazon's EC2 Cloud service for faster computation. A better integration of EAs with Cloud models has already been pursued by some researchers. Two noteworthy examples are the Offspring framework by Vecchiola et al. [45] and the FlexGP system developed by Sherry et al. [39]. Offspring runs a multiobjective EA that is executed on Aneka Enterprise Clouds using a simple distribution logic that makes the use of the cloud transparent for the user, developed on top of the task model with a plug-in architecture. FlexGP is probably the first large scale GP system that runs on the cloud, using an Island model approach and implemented over Amazon EC2 with a socket-based client-server architecture. Other recent contributions include several tools and libraries designed to distribute an EA over the Cloud using a global queue of tasks and a Map-Reduce implementation [39].

All of these works show great promise in extending EAs towards modern computing models. However, they focus on exploiting current distributed computing technologies to enhance EA performance, while maintaining the same basic evolutionary model. In other words, in terms of the search the EA might perform the exact same process, even if it is done more efficiently through distributed computing technologies. On the other hand, the current work deals with a novel EA model introduced by Pool-EAs. What is particularly interesting in a Pool-EA, is that they are explicitly designed to exploit a distributed computing framework, while also employing an EA model that incorporates non-traditional search dynamics. In a Pool-EA, the evolving population is stored in a centralized repository or store, while distributed clients asynchronously extract a subset of individuals and return a new subset of individuals after performing the actual search operators. Pool-EAs are related to more general systems, such as the A-Teams system proposed in [40], a problem-solving multi-agent architecture based on a strongly cyclic network. G. Roy et al. [36] also developed a multi-threaded system with a shared memory architecture that is executed within a distributed environment, where the evolving population of a GA is stored in a centralized pool or

[1] http://www.nutonian.com/

database. Another example is developed by Bollini and Piastra [5], who proposed a system that decouples population storage from the actual evolutionary operations.

Pool-EAs are closely related with Island-Model EAs (IMEA) [7] with several noteworthy differences [30]. In an IMEA, each island represents a partially independent evolutionary process, with a periodic exchange of individuals following a synchronized and coordinated protocol, using a fixed topological structure. On the other hand, a Pool-EA does not employ such a coordinated interaction scheme, workers perform an evolutionary process using a sample of individuals taken from the central pool. Moreover, the manner in which workers interact with the central pool is completely random and asynchronous. These conceptual difference illustrate how a Pool-EA offers a much simpler approach towards the development of a distributed evolutionary search, eliminating some of the parameters or design choices that are not easy to determine in an IMEA application (such as the number of islands, migration strategies and communication topologies). In other words, a Pool-EA allows for the deployment of an ad-hoc distributed EA, where connected clients can connect and disconnect from the EA process without strict prior restrictions or system configuration. However, whether or not these differences provide any performance advantages can be seen as an empirical question, one that is considered in the experimental sections of this paper. Moreover, even if it does not, this paper attempts to determine the performance of a pool-based system when it is the only, or the best, alternative to implement a distributed evolutionary algorithm.

The most complete Pool-EA model proposed thus far is the SofEA algorithm by Merelo et al. [29, 31, 32]. It is an evolutionary algorithm mapped to a central CouchDB object store; CouchDB is one of several such systems available on the market; its main advantage over others is its massive concurrency due to its Erlang base and its API based in REST and JavaScript. SofEA provides an asynchronous and distributed search process, where the four main evolutionary operators are completely decoupled from the evolving population; these are: Initialization, Evaluation, Reproduction and Elimination. The last three processes can be executed in any order and in any given time step, and more than a single Evaluator and Reproducer can be executed concurrently. EvoSpace shares several similarities with SofEA, but it also has several noteworthy differences. First, while SofEA also presents a distributed and asynchronous model, the evolutionary process is still carried out on the central repository. On the other hand, in EvoSpace evolution is carried out locally on each client, as if small sub-populations are periodically isolated, evolved and then returned to the central store. Second, while both systems attempt to decouple population storage from the search operations, SofEA maintains a fine-grained control of each individual within the population that limits the ability of each to interact with remote clients. For instance, once an individual is evaluated in SofEA, its fitness cannot be reassigned based on a future evaluation, this limitation is particularly important in the development of interactive systems where several users could provide input regarding the quality of a solution [20]. Thirdly, since SofEA considers each individual as unique, it does not allow duplicates to appear within the evolving population, something that is useful for diversity preservation but that might curtail the exploitation capability of the EA, since multiple copies of a particularly high-quality solution cannot be present. Furthermore, if an individual dies (or is eliminated), then another copy of it cannot be reintroduced into the population, even if it might be useful at a later stage of the search. Finally, by using the individual chromosome as ID, it appears problematic to extend SofEAs to more complex representations such as GP, an extension that is easily accomplished with EvoSpace.

3 The EvoSpace Model

The ESM is a conceptual framework for the development of Pool-based EAs. Figure 1 shows a graphical illustration of the main components and dataflow within the ESM. The main components are: the EvoStore population container, remote clients called EvoWorkers, and the stored objects or Individuals of an EA search. Each of these components are defined in the following subsections.

3.1 The EvoStore Container

The EvoStore is an associatively addressed memory space shared by several processes following the

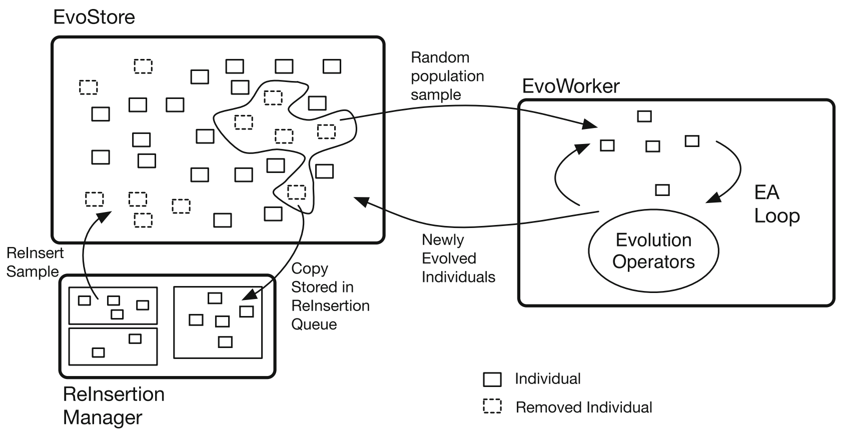


Fig. 1 Main components and dataflow within the EvoSpace model

principles of the tuple-space model, which can be described as a distributed shared memory (DSM) abstraction, organized as a bag of tuples. A tuple t is the basic tuple space element, composed by one or more fields and corresponding values. In this model, the basic operations that a process can perform is to insert or withdraw tuples from the tuple space. An EvoStore provides a set of interface methods that operate over a set of objects ES, which can be seen as an independent EA population. If multiple populations are needed, each one must have an index as in ES_i. For simplicity in this description, we assume an EvoStore has a single population ES. Objects in ES represent individuals in the population (these are defined in the next subsection), they can be removed and replaced from ES using a specified set of methods. However, the ESM is different from other tuple space implementations in the sense that retrieving and reading objects from ES are defined as random operations. Retrieving a single objects is not of high interest when accessing the EvoStore, instead samples of the population are desired, a key difference with the SofEA approach [29, 31, 32]. The methods provided by the EvoStore container are the following:

- **Read(n):** This method returns a random set A of objects from ES, with $|A| = n$ and $A \subset ES$, if $n < |ES|$, the method returns ES otherwise. When using this method, objects are not removed from ES meaning that they are intended for read-only operations. This method is useful to analyze the evolving population without interfering with the search, to compute performance measures online.
- **Take(n):** Returns a random set A, following the same logic used for $Read(n)$. However, in this case the set A is removed from ES, the contents of the EvoStore are then given by $ES = ES \setminus A$. We define A_i as the set returned by the $i - thTake()$ operation. The objects taken are also copied to a new set S_i of removed samples and stored within a temporary collection $\mathcal{S}$ on the server, implemented as a priority queue. Sets $S_i \in \mathcal{S}$ can then be reinserted to ES if necessary.
- **ReInsert(i):** This method is used to reinsert the subset of elements removed by the $i - thTake()$ operation, such that the contents of EvoSpace are now $ES \cup S_i$ if $S_i \in \mathcal{S}$, ES is left unchanged otherwise. In other words, the method ReInserts previously taken samples from the population store. The ReInsert method can be triggered when a worker loses its connection to the server or when the population size gets below a threshold, this prevents the starvation of the population pool and also compensates for lost work. Other distributed algorithms normally use a random insertion technique, but this might negatively impact the search process if good solutions are lost.
- **Insert(A):** This method represents the union operation $ES \cup A$, where A is a set of individuals. This method is normally used to initialize the population, with a randomly generated set of individuals. This differs from ReInsert, which introduces previously taken individuals. Section 6, analyzes the

effects of using Insert (with a random set of individuals) instead of ReInsert (with individuals from a previously unreturned sample) when the population pool is starved.

– **Replace(A,i):** Similar to $Insert()$, however set A should be understood as a replacement for some $S_i \in \mathcal{S}$, hence $|A| = |S_i|$, but the objects in A can be different (evolved) objects from those in S_i. Moreover, if S_i exists it is removed from $\mathcal{S}$. However, if S_i does not exist this means that a $ReInsert(i)$ operation preceded it, this increases the size of ES.
– **Remove(A):** This method removes all of the objects in A that are also in ES, in such a way that the contents of EvoSpace are now set to $ES \setminus (A \cap ES)$.

3.2 Individuals

As stated above, the objects stored in ES represent individuals for an EA. Explicitly, the objects in ES are stored as dictionaries, an abstract data type that represents a collection of unique keys and values with a one to one association. Objects are described by the following basic fields: an **id** string that represents a unique identifier for each object; a **chromosome** string, which depends on the EA and the representation used; a **fitness** dictionary for each individual, which allows the system to store multiple fitness values for multiobjective or dynamic scenarios; a **parents** dictionary with identifiers of the individual(s) from which it was produced. Moreover, this representation can be extended according to the requirements of each implementation.

3.3 EvoWorkers

EvoWorkers are autonomous computational entities that are executed on client machines. They only communicate with the EvoStore but not with each other. Communication is carried out by message passing, as described in Algorithm 1. Each EvoWorker runs a local algorithm that executes the main code of an EA search. The **EvoWorker** process is straightforward, it requests a set of objects A_j taken from ES. Afterwards, locally the $Evolve()$ function is called where the actual evolutionary cycle is performed. In this scenario, A_j can be seen as a local population on which evolution is carried out for g generations. The result of this evolution is then returned back into ES; then, the EvoWorker can request a new set and repeat the process. Otherwise, each EvoWorker could specialize on a particular part of the evolutionary process, such as selection, evaluation or genetic variation, an approach taken by SofEA [31].

In summary, the core ESM elements are the EvoStore and EvoWorkers; however, additional components must be defined to implement a complete Pool-EA, but these will be constrained by the structure and functionality of the core elements. Moreover, all other components have to be designed in accordance with the particular type of Pool-EA to be implemented. Nonetheless, these are still abstract components, not tied to a particular implementation, as described next.

3.4 The EvoSpace Server Components

The EvoSpace Server employs a client-server architecture using at its core an EvoStore container. On the server side, a process called **EvoSpaceServer** is executed, which creates and activates a new EvoStore container object and waits for requests to execute interface methods; see Algorithm 2. Additionally, on the server two more processes are executed, these are: **InitializePopulation** and **ReInsertionManager**; see algorithms 3 and 4. **InitializePopulation** is executed once, its goal is obvious, initialize the population by adding $popsize$ random individuals by calling the $Insert(A)$ method . The function that creates new individuals depends on the problem and representation used. **ReInsertionManager** is used as a fail-safe process that periodically checks (every wt seconds) if the size of the population in ES falls below a certain threshold min_p, what is known as pool or population starving. When this scenario occurs, then rn subsets $S_i \in \mathcal{S}$ are reinserted into ES using the $ReInsert(i)$ method. Moreover, the **ReInsertionManager** could also be configured so that $ReInsert(i)$ is called when a particular $EvoWorker$ has lost a connection with the server, after waiting for a maximum amount of time for a return (a timeout), in this way recovering the specific population sample taken by the EvoWorker; however, this implementation was not tested in the experiments presented in this paper. Nonetheless, two different strategies are compared in Section 6, the first one is Algorithm 4, and a second version that instead of using the $ReInsert(i)$ method, generates a random replacement for S_i and inserts t into ES by calling $Insert(A)$ (see Fig. 1).

Algorithm 1 The client-side **EvoWorker** process.

```
Require: $EvoSpace \leftarrow$ Reference to an Evospace Server
Require: $n \leftarrow$ sample size
Require: $rt \leftarrow$ retry time
Require: $g \leftarrow$ number of generations
  while EvoSpace.ES.active do
    $S_i \leftarrow ES.Take(n)$
    if $A_i$ is not null then
      $Evolve(A, g)$
      $EvoSpace.ES.Replace(A_i)$
    else
      $wait(rt)$
    end if
  end while
```

Algorithm 2 The server-side **EvoSpaceServer** process.

```
  $EvoSpace \leftarrow$ new EvoSpace
  $EvoSpace.active \leftarrow$ true
  while $EvoSpace.active$ do
    read method
    return $EvoSpace.method()$
  end while
```

Algorithm 3 The server-side **InitializePopulation** process.

```
Require: $EvoSpace \leftarrow$ Reference to an Evospace Server
Require: $popsize \leftarrow$ Number of individuals
  $j \leftarrow 0$
  for $j < popsize$ do
    $ind \leftarrow$ new individual() {Problem dependent}
    $EvoSpace.ES.Insert(\{i\})$
    j++
  end for
```

Algorithm 4 The server-side **ReInsertionManager** process.

```
Require: $min_p \leftarrow$ population size threshold
Require: $rn \leftarrow$ number of samples to re-insert
Require: $EvoSpace \leftarrow$ Reference to an Evospace Server
Require: $wt \leftarrow$ wait interval
  while $EvoSpace.active$ do
    if $|EvoSpace.ES| \leq min_p$ then
      $EvoSpace.ReInsert(rn)$
    end if
    $wait(wt)$
  end while
```

4 Reference Implementation: EvoSpace-Py

In this section a reference implementation of an ESM-based Pool-EA is presented. The system is called EvoSpace-py since it was implemented using the Python language. In the proposed implementation, individuals are stored in-memory, using the Redis key-value database redis.io. Redis was chosen over a SQL-based management system, or other non-SQL

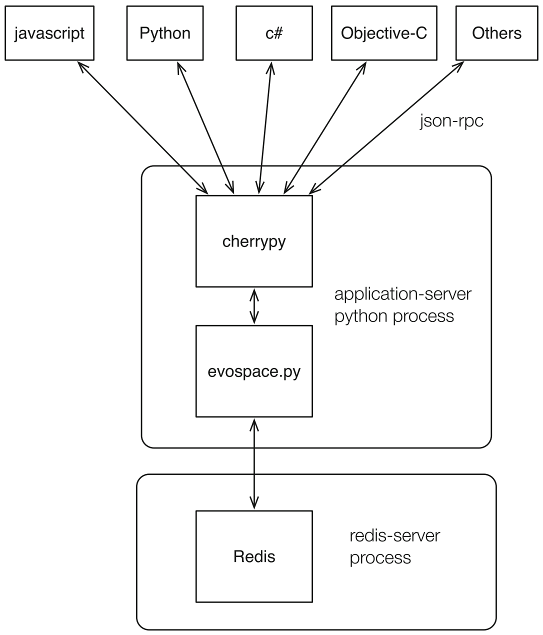


Fig. 2 The EvoSpace-py implementation architecture

alternatives, because it provides a hash based implementation of sets and queues which are natural data structures for the EvoStore container. For instance, selecting a random key from a set has a complexity of O(1). The logic of the EvoStore and EvoSpace server is implemented as a Python module exposed as a JSON-RPC Service using Cherrypy and Django HTTP frameworks. In this way, the developed EvoStore server can interact with any language supporting json-rpc or Ajax requests. The EvoStore module, EvoSpace server and EvoWorker implementations in JavaScript and Python are available with a Simplified BSD License from http://github.com/mariosky/EvoSpace.

On the other hand, EvoWorkers must implement the genetic operators for a particular EA. Given that EvoSpace-py is implemented as a Python module, a simple way to implement EvoWorkers is by using the basic non-distributed GA found in the Distributed Evolutionary Algorithms in Python (DEAP) library [16]. However, three methods were added to the local DEAP algorithm: $getSample()$ and $setSample()$; and another for the initialization of the population using DEAPs own methods. For instance, to generate the initial population DEAP's $initialize()$ method is called and the generated population is sent to

EvoSpace, then for each generation $getSample()$ function is called, this executes the EvoStore $Take()$ method, and the received sample is used to create a DEAP population which is then evolved for a predefined number of generations. Finally, the $setSample()$ function converts the DEAP population to a JSON object and returns it to the EvoStore through the $Replace()$ method.

4.1 EvoSpace-py in a Cloud Architecture

The first version of EvoSpace-py was implemented as a multi-process application running in a single computer; this implementation was used to run the first set of experiments described in Section 5. However, a cloud based implementation was needed to further evaluate the ESM. This section describes how EvoSpace-py can be configured to run on a cloud architecture using two Platform as a Service (PaaS) components, Heroku for the EvoSpace Server and PiCloud for simulating EvoWorkers; a schematic view of the cloud architecture is shown in Fig. 3.

Heroku (http://heroku.com) is a multi-language PaaS, supporting among others Ruby, Python and Java applications. The basic unit of composition on Heroku is a lightweight container running a single user-specified process. These containers, which they call *dynos*, can include web (only these can receive `HTTP` requests) and worker processes (for instance processes used for database backups or task queues). These process types are the prototypes from which one or more dynos can be instantiated; if the number of requests to the server increases then more instances can be assigned on-the-fly. In our case, our CherryPy web application server runs in one web process, when the number of workers was increased we added more dynos (instances) of the CherryPy process. This model is very different from a Virtual Private Server (VPS) where users pay for the whole server; in a process based model, users pay only for the processes they need; being a *freemium* model means also that, if a minimum level of resources is not exceeded, it can be used for free. Once deployed the web process can be scaled up by assigning more dynos; for instance, in the more demanding experiments the web process was scaled to 20 dynos.

4.2 EvoWorkers as PiCloud Jobs

PiCloud is a PaaS, with deep Python integration, such that functions are transparently uploaded to PiCLoud's servers as units of computational work they call jobs. Each job is added to a queue, and when there is a computing core available, the job is assigned to it. PiCloud has several options of computing resources they call cores, for the experiments type c1 and c2 Real Time cores where used; c1 cores have 1 Compute Unit and low I/O Performance, on the other hand c2 cores have 2.5 Compute Units and moderate I/O Performance.

5 Experiments and Results

The goal of this section is to evaluate the ESM and the proposed EvoSpace-py implementation, using

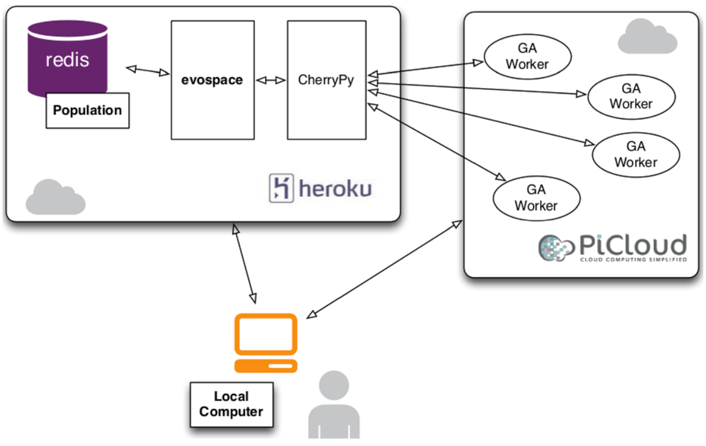


Fig. 3 EvoSpace-py cloud-based architecture

well-known benchmarks for GAs, the OneMax problem, the K-trap problem [42] and the P-Peaks problem [11]. In particular, two different versions of EvoSpace-py are evaluated, a local multi-process implementation and a cloud-based implementation, using three experimental setups. In the first experiment, the ESM is compared with the IMEA using the OneMax problem, to evaluate how it compares to a closely related EA model that is also amenable to distributed computing frameworks. In the second experiment, the ESM is evaluated based on its ability to find local optima on the deceptive K-trap problem, comparing its performance that of a canonical GA. Finally, in the third experimental setup the speedup offered by the cloud-based implementation is evaluated given the communication overhead that the distributed approach requires, comparing it with a local execution. Experiment A are done on an Intel Xeon E5 with 10MB of L3 cache and Turbo Boost up to 3.9GHz, 12GB of 1866MHz DDR3 ECC memory running Mac OS X 10.9 and the Python interpreter version 2.7.2 for 64-bit architectures. The specifications for the local computers used in Experiments B and C are a 2.2 Ghz Intel Core i7 processor, 16 GB of 1333 DDR3 memory, and Mac OS X 10.7.5.

5.1 Experiment A: Comparison with IMEA Using OneMax

These experiments focus on comparing EvoSpace with an IMEA using the classic OneMax problem. In particular, the goal is to evaluate search efficiency and the computational effort of each of these strategies. To do so, a 128-bit OneMax problem is chosen, using a binary-coded GA with one point mutation and two point crossover. The population is set to the minimum size that guarantees that a solution will be found in over 90% of the runs following the strategy described in [28], additional configuration details are given in Table 1. EvoSpace and the IMEA are executed locally, for the latter the reference implementation provided by DEAP is used. For EvoSpace, the problem is tested with 2,3,4 and 6 distributed EvoWorkers, and for the IMEA the same number of demes, or islands, are used. In each case, a different population (or deme) size is used, or in the case of EvoSpace a different sample size, following [28].

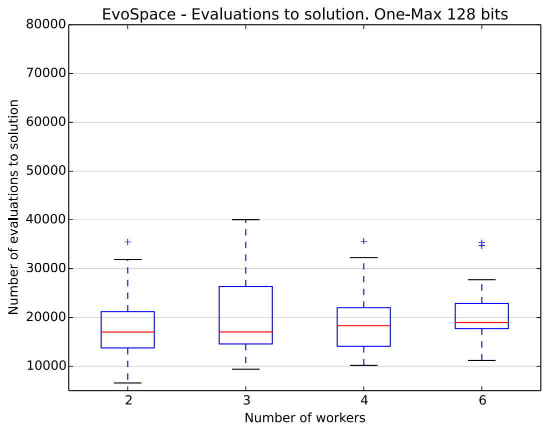
Figure 4 presents a comparison based on the total number of fitness function evaluations required

Table 1 Parameters for EvoSpace and the IMEA on the OneMax problem

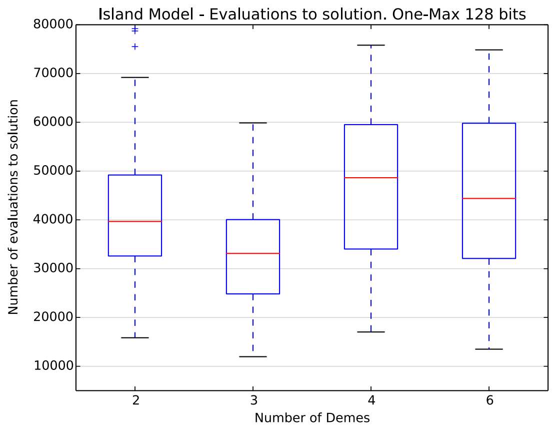
EvoSpace	
Population size	64,64,64,52
Sample Size	24,18,12,8
Worker generations	30
IMEA	
Deme size	80,40,38,25
Generations	30
Migration rate	5
Shared Parameters	
Crossover prob.	0.5
Mutation prob.	0.2
Tournament selection	size 4

to find a solution and the total time to solution. Figures 4a and 4b show the total number of evaluations for EvoSpace and IMEA respectively, and Figs. 4c and 4d show the same for total time to solution. First, it is clearly shown that EvoSpace performs a more efficient search that the IMEA, requiring about half the total number of fitness evaluations in all cases. Moreover, with EvoSpace the number of fitness evaluations seems to be independent of the number of EvoWorkers that participate in the search, since in all cases the median number of evaluations is below 20K. Moreover, this is validated with a KruskalWallis statistical test, it gives a $p_{value} = 0.423$ which does not reject the null hypothesis that the different groups shown in Fig. 4a come from the same distribution. On the other hand, the IMEA shows greater variability, with median values varying from around 35k to as much as 50k depending on the number of demes, without a clear trend. In this case, the KruskalWallis test returned a $p_{value} = 0.009$, rejecting the null hypothesis and suggesting that the number of function evaluations depends on the number of dems for an IMEA.

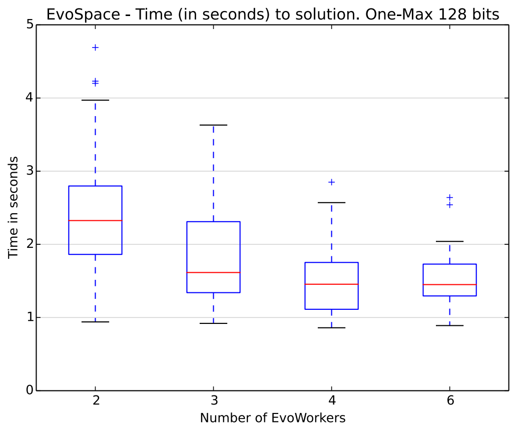
On the other hand, regarding time to solution, it seems clear that with more workers EvoSpace performs better, with median results of 2.3s with 2 EvoWorkers down to 1.5 with six EvoWorkers. The IMEA, however, varies between 0.7s and 1.1s, without any clear improvement as the number of demes increases. These results are expected, given that the IMEA is executed within the same DEAP process,



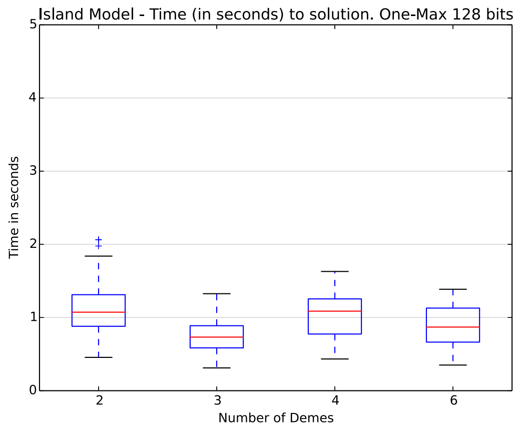
(a) EvoSpace



(b) IMEA



(c) EvoSpace



(d) IMEA

Fig. 4 Comparison of EvoSpace with an IMEA: **a, b** Number of fitness function evaluations; **c, d** Total time to solution

while EvoSpace is incurring a communication overhead between Redis, JSON and DEAP. However, in a distributed network such overheads would be unavoidable for any model, and in real-world scenarios fitness evaluations tend to be the most severe bottleneck. Therefore, these results suggest that EvoSpace can perform a more efficient search process that an IMEA.

5.2 Experiment B: K-trap Function

For these experiments the K-trap function is used to test the ESM, a problem that presents a multi-modal and deceptive fitness landscape. Table 2 summarizes the different experimental configurations tested, based on the K value, number of EvoWorkers, the sample size taken by each worker and the chromosome length. A bit-string representation is used, and each EvoWorker performs 100 total generations on each sample, using one-point crossover (with crossover probability set to 1) and one-point mutation (mutation probability set to 0.06). The number of individuals in the EvoStore is set to 1024 for 4-trap experiments and to 4096 for 5-trap, and the maximum number of total samples that can be taken from the EvoStore in each run is set to 1000. For comparison, a standard GA is applied to each benchmark problem. For

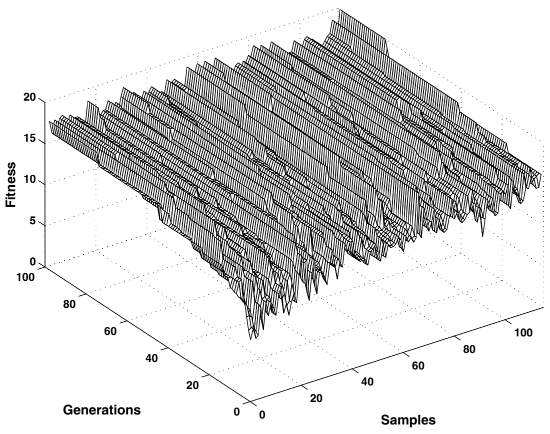
Table 2 Different experimental configurations used to test the performance of EvoSpace

Experiment	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
K-trap	4	4	4	4	4	4	4	4	4	5	5	5	5	5	5
EvoWorkers	1	1	4	4	8	8	16	16	32	1	4	8	16	32	40
Sample size	32	64	32	64	32	64	32	64	32	64	64	64	64	64	64
Chromosome	40	40	40	40	40	40	40	40	40	50	50	50	50	50	50

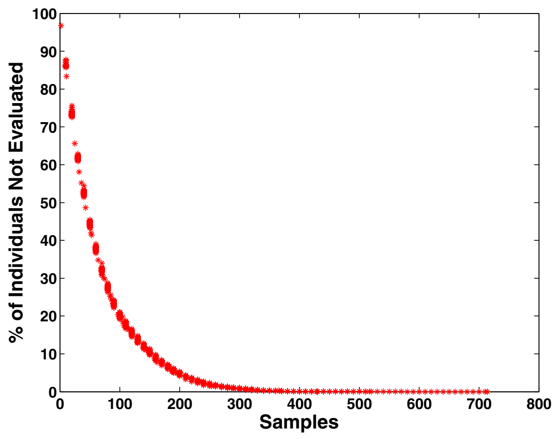
the 4-trap problem the maximum number of generations is set to 4000, and for the 5-trap problem it is set to 1000. These values limit the maximum number of function evaluations that can be performed, in fact these values were chosen so the limit is equivalent to the maximum number of function evaluations from all of the EvoSpace runs; however, most EvoSpace runs required much less function evaluations than these maximum values. In total, 50 runs were performed for each experimental configuration.

Figure 5a depicts how fitness evolves over all of the samples taken from EvoSpace. This figure shows the evolution of best-fitness for a single run of experiment K in Table 2; the analysis focuses on a single run instead of the mean of all runs to emphasize the local dynamics of the evolutionary process. The plot shows how fitness evolved on each EvoWorker that participated in the search. Evolution of fitness is organized based on the two temporal axis of the horizontal plane, one corresponds with the sample number, independent of which EvoWorker took the sample, and the other corresponds to the generations of the local evolutionary process executed on the EvoWorker. In other words, these plots provide a collective view of the evolutionary process from the perspective of all EvoWorkers. Since the global optimum is a fitness value of 20, we can see that the evolution on the last sample taken from EvoSpace reaches the global optimum; also notice the low fitness value for the first few samples taken in the initial generations.

EvoSpace outperforms the standard GA on both tests, with a substantial increase in the number of optima found. In eleven experiments EvoSpace found the optimal solution in all runs, in the other four cases (experiment A,D,E and H) it does no worse than 48 total optima found. On the other hand, the sequential GA only finds 34 optima on the 5-trap problem and 29 on the 4-trap case. These results were not expected, since the EvoSpace algorithm is using the same representation and search operators (mutation and crossover) than the standard GA. These results suggest that the population dynamics induced by the



(a) Evolution of Fitness



(b) Evaluated Individuals

Fig. 5 These plots summarize the results for Experiment K: **a** evolution of fitness for a single run, the plot shows how fitness evolves for each sample taken by the EvoWorkers; **b** scatter plot, considering all runs, of the percentage of non-evaluated individuals

ESM might improve the quality of the results, while otherwise using a basic representation and genetic operators.

Since every EvoWorker takes a random sample of individuals, one concern might be that some individuals of the initial population might not be chosen and evaluated wasting valuable genetic material. Figure 5b shows a scatter plot of all of the runs for experiment K, depicting the percentage of individuals that have not yet been evaluated; here it is important to remember that some runs required more samples than others. The figure clearly shows that the percentage individuals not evaluated within EvoSpace quickly decreases as more samples are taken.

Finally, a comparison of the computational effort required in each experiment is given in Fig. 6, which shows boxplots of all runs in each experiment. Figure 6a plots the total number of individuals evaluated in each run, which is similar in all experiments and consistent with the results from Experiment A; while Fig. 6b compares the total run time in seconds. These figures show that run time is significantly reduced as the number of EvoWorkers increases.

5.3 Experiment C: P-Peaks

The P-Peaks problem was chosen because the problem and consequently the computing resources needed for the search can be appropriately scaled. Proposed by De Jong et al. in [11], as a generalization of the version in [12], a P-Peaks instance is created by generating a set of P random N-bit strings, which represent the location of the P peaks in the space. To evaluate an arbitrary bit string **x** first locate the nearest peak (in Hamming space). Then the fitness of the bit string is the number of bits the string has in common with that nearest peak, divided by N. The optimum fitness for an individual is 1, and is computed by

$$f_{P-PEAKS}(\mathbf{x}) = \frac{1}{N} \max_{i=1}^{P} \{N - hamming(\mathbf{x}, Peak_i)\} . \tag{1}$$

A large number of peaks induces a time-consuming search, since evaluating every string is computationally expensive; this is convenient since in order to justify a distributed EA fitness computation has to be significantly larger than the associated network latency (otherwise, it would always be faster to have a single-processor version). For this work, the experiment is setup with $P = 256$ peaks and $N = 512$ bits, a configuration that requires considerable computational time for fitness evaluation, and 30 runs are performed. Regarding algorithm parameters these are summarized in Table 3, in particular notice that these experiments use PiCloud's c2 Real Time cores.

As a baseline execution, the experiment was also conducted on a local computer. In this setting, the

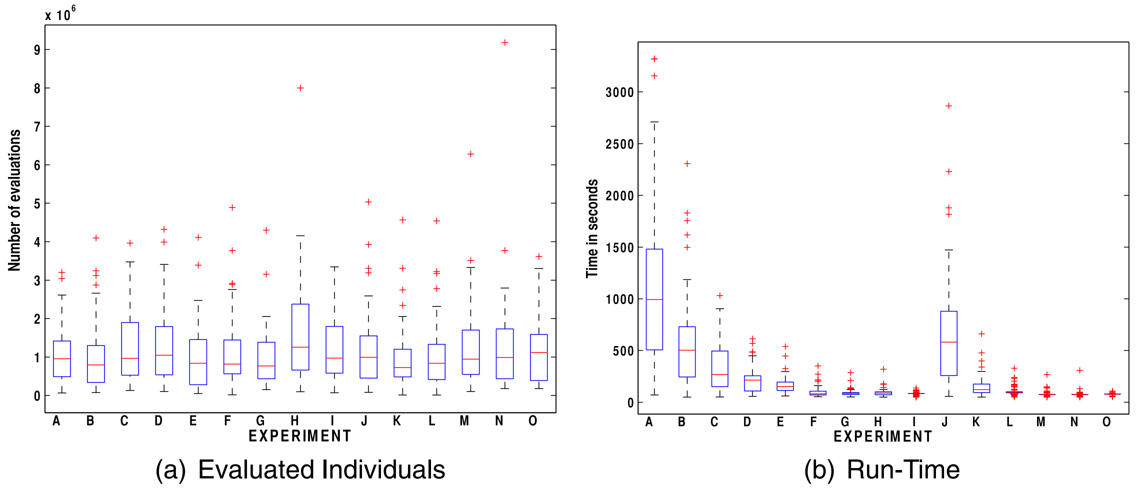


Fig. 6 The plots show a performance summary for all of the experiments, based on: **a** number of evaluated individuals over all runs; and **b** total run-time

Table 3 GA and EvoWorker parameters for Experiment C

GA Parameters	
Tournament size	4
Crossover rate	0.85
Population Size	512
Mutation probability	0.5
Independent bit flip probability	0.02
EvoWorker Parameters	
Sample Size	16
Generations	128
Variable Parameters	
PiCloud Worker Type	c1 Realtime, c2 Unreliable Realtime
Number of Workers	2,4,8,16,28
Number of Executions	30

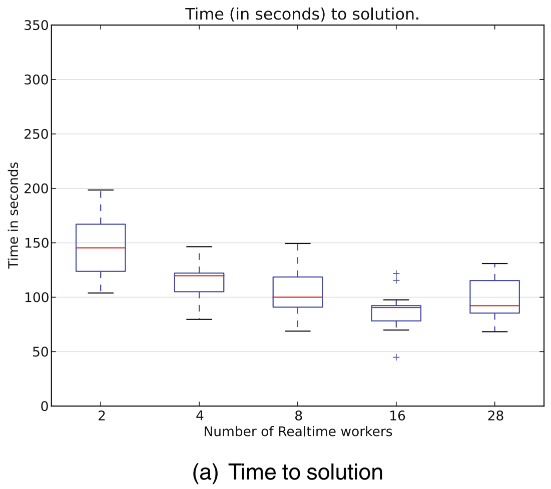
problem required considerable computational time: each run took an average of 1567.36 seconds to find the optimal solution. The execution used a single core and CPU activity remained low for the whole length of the experiment. On the other hand, the parallel execution time was significantly lower even when only two EvoWorkers were used, clocking in at less than 180s even in the worst cases.

Average times for the configuration using Realtime cores in PiCloud are presented in Fig. 7a. It can be seen that incrementing the number of workers reduced the time to solution, but only up to 16 EvoWorkers, however with 28 EvoWorkers the median time to solution did not improve. There seems to be a point at which the overall speedup of the distributed search levels off in this experimental setup.

For problem domains where fitness evaluation of individuals is not demanding, the added overhead of communication between the EvoStore and the EvoWorkers can become a concern. However, our experiments suggest that this cost is practically negligible. Figure 7b presents boxplots of the time required to perform the three main EvoSpace-py functions on the EvoWorkers, computed over all of the 30 runs using 28 Realtime PiCloud workers. It is clear that almost all of the computational time is consumed by the $Evolve()$ function which actually performs the search operations, while the cost of taking a and returning a sample is relatively low.

6 Overcoming Problems and Limitations

The previous section showed how the ESM can be used to implement a distributed and asynchronous Pool-EA based on the EvoSpace-py implementation, with strong results. However, as stated before, there are possible downsides to using a Pool-EA, some of which are directly addressed by the underlying ESM. However, some issues persist, particularly the critical



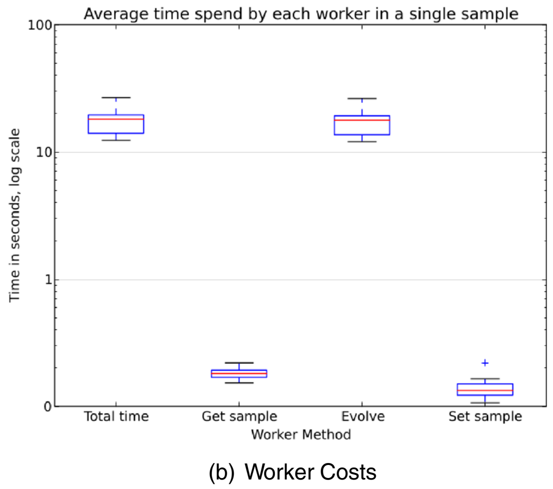


Fig. 7 Evaluation on the P-Peaks problem: **a** Time required to solution; and **b** Number of evaluations to solution

problem of possible lost work due to the unreliable connection of EvoWorkers, and the tedious problem of algorithm parametrization, a common issue with almost all EAs that is severely amplified in a Pool-EA.

6.1 Unreliable Workers

In this section, the effect of node unavailability in an EvoSpace Pool-EA is assessed. The ESM contrasts with the use of a global queue of tasks and implementations of Map-Reduce algorithms, such as in [14], with several benefits relevant to concurrency control and workload distribution. EvoWorkers are expected to be unreliable, since they can loose a connection or could simply shut down or be removed from the client machine. When an EvoWorker is lost, so are the individuals pulled from the population store. Depending on the type of algorithm that is executed, the loss of these samples could have a high performance cost. As stated before, to address this problem the ESM uses a reinsertion algorithm that also prevents the starvation of the population pool. Other pool based algorithms normally use a random insertion technique, but this might negatively impact the search process.

Hence, the goal of this section is to evaluate the effect the reinsertion algorithm has on the total running time and number of evaluations of a GA, using the P-Peaks problem described above with the same parametrization described in Table 4; in this case c1 Realtime cores are used, which are less efficient than the c2 cores used in Section 5.3. In particular, two distinct approaches are evaluated: (a) reinserting previously taken individuals, at the cost of keeping copies of samples; and (b) inserting randomly generated individuals, which has the added effect of increasing diversity within the population.

The algorithm stops when reaching the optimum value, or when all EvoWorkers pulled 100 samples. To simulate unreliable workers, each was assigned a return sample probability. In the experiments the lower probability was a 30 % chance of an EvoWorker returning a sample or an EvoWorker failing 70 % of the time; other return sample probabilities where 50 %, 70 % and 90 %. Here, it might be said that workers with a low return probability (say 30 %) should in fact be avoided, and a proper strategy would be to eliminate those workers from the evolutionary process. However, we consider a scenario were the reliability of an Evoworker is not known a priori, their reliability can only be assessed after an experiment is conducted (or during). Over time, however, it might be possible to build historic profiles of participating EvoWorkers at take appropriate steps to improve performance when a highly unreliable worker is detected, but incorporating such strategies is left as future work.

Experiments where carried out using a total of 4, 8 and 16 EvoWorkers. Although supported by EvoSpace, time intervals were not chosen as triggers to feed the population with new individuals, population size was used instead. The population size is a better threshold as it is more critical to a GA performance, for instance in experiments where the population remaining in the pool was near starvation, the time to completion increased. For these experiments, the insertion of individuals was triggered when less than 128 individuals remain in the EvoStore; the number of individuals fed to the population was 128, or 8 complete samples, when the reinsertion algorithm was used.

Figure 8a shows the time required to solution when using four EvoWorkers. For a population of 512 individuals and a sample size of 16, there is no difference in the time required to solution for percentages of 50 % and above. Both reinsertion algorithms had comparable times. For 30 percent, both approaches had a slight increase in time. For 8 EvoWorkers, shown in Fig. 8b, there was a marginal decrease in overall time; and results where similar to those found in the experiments with 4 EvoWorkers. Figure 8c shows results for 16 EvoWorkers, when there was only a 30 % chance of returning a sample the rate of reinsertion was high, this produced one reinsertion event approximately once every 35 samples, considering that 8 samples were reinserted at every such event. In this case, the insertion of random individuals resulted in a higher time to solution.

In summary, the results suggest that the reinsertion algorithm is better for situations when starvation of the pool is common. On the other hand, inserting random individuals is not detrimental when there are other evolved individuals within the pool, but when the remaining pool mainly consists of random individuals, new samples pulled by EvoWorkers need to start the search from scratch. Therefore, if only a small number of samples are returned to the pool, the work needed to reach the optimum is increased. Figure 8d also shows the number of evaluations needed to reach an optimum for 16 EvoWorkers. Figures 8e and 8f

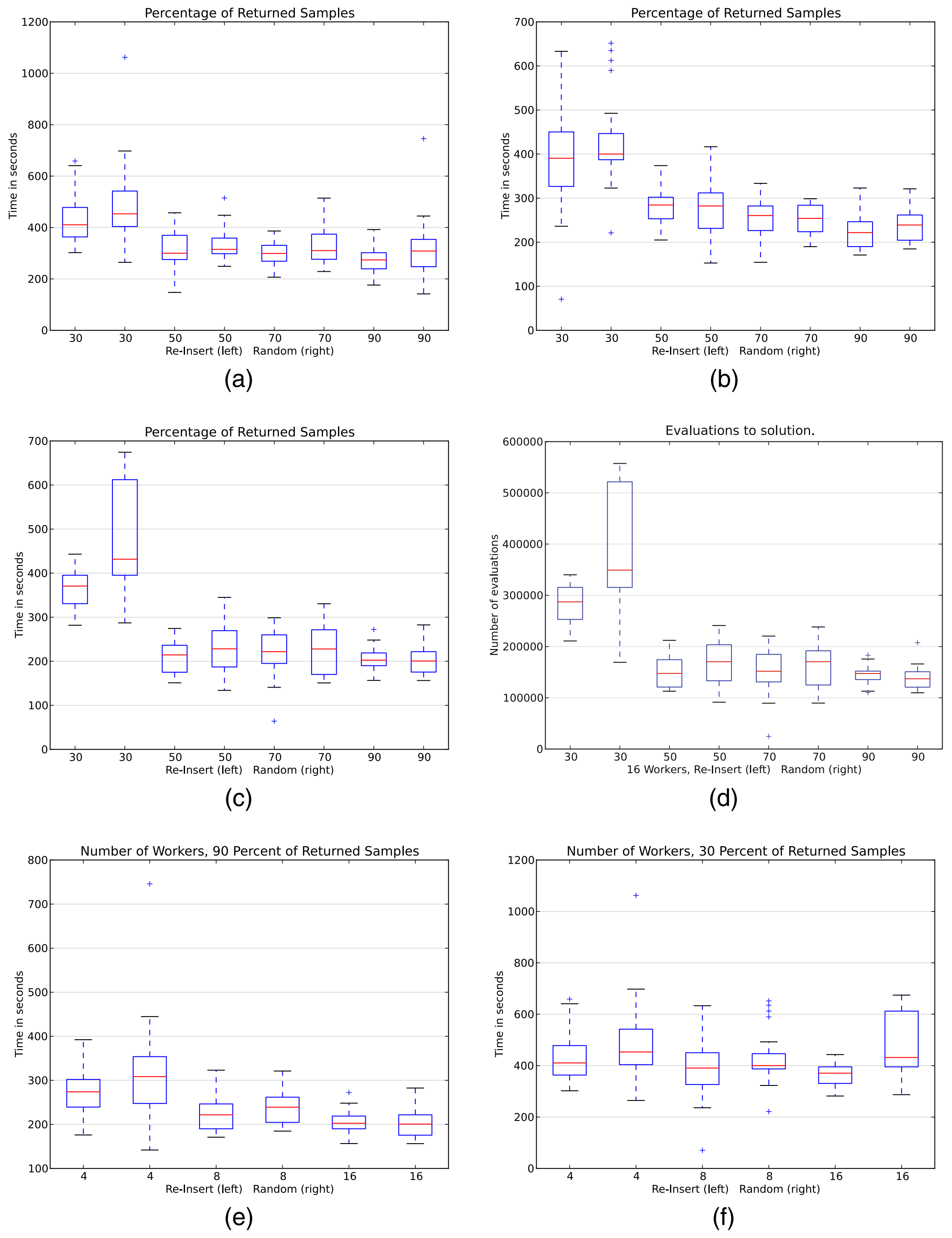


Fig. 8 Unreliable workers: **a** Time to solution, 4 Workers; **b** Time to solution, 8 EvoWorkers; **c** Time to solution, 16 EvoWorkers; **d** Number of evaluations, 16 EvoWorkers; **e** Time to solution, 90 % returned samples; **f** Time to solution, 30 % returned samples

show the time required to solution for 30 % and 90 % of returned samples, two extreme cases. For 90 % both algorithms had similar speedups as the number of EvoWorkers increases. Conversely, for a return probability of 30 % there is practically no speedup at all with more EvoWorkers. The reinsert algorithm, however, does show a lower median total time compared with the random strategy, particularly with 4 and 16 EvoWorkers.

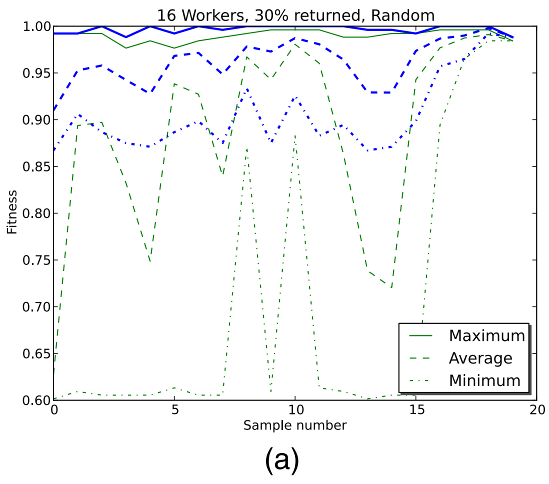
Fitness with respect to time was measured as the average from each consecutive sample pulled by each worker. For each sample, the average fitness was measured at the start and at the end of the local evolution. Also the minimum and maximum fitness values at the start and finish was recorded. Figure 9a shows the evolution of fitness with the random insertion algorithm, where the initial fitness drops at certain points when random insertion occurs, while average final fitness is also compromised. Figure 9b shows results for the reinsertion algorithm, with more characteristic convergence curves without substantial fitness drops.

6.2 Parametrization

In general, EAs are sometimes criticized by the large number of parameters that need empirical tuning in order to get them to work properly or reach the required performance or require additional heuristic processes to be included into the search to adjust the parameters automatically [25, 27]. In the case of Pool-EAs, this issue is magnified since the underlying system architecture adds several degrees of freedom to the search process, with unknown interactions. This problem is of particular importance in real-world scenarios, where there might be little prior insights regarding what could be the best configuration for an EA tool, especially if the intent is to use it as a black-box optimizer; a comprehensive survey on this topic is given in [27].

A noteworthy contribution is made by Cantú Paz [7], who addresses the problem of deriving theoretical models of the effects of parameters related to population size and migration in Island-Model EAs (IMEA). However, it does not cover the effects of all possible parameters, or the intricacies of a Pool-EA algorithm. Therefore, some of the well-known insights derived from IMEA research (regarding, for example, migration policies) are not necessarily relevant in the Pool-EA framework.

Therefore, the recently proposed approach called Randomized Parameter Setting Strategy (RPSS) [23, 41] is tested with EvoSpace in this section. The idea behind RPSS is that in a distributed EA, algorithm parametrization may be completely skipped to conduct a successful search. The first works with RPSS focused on an IMEA model [23, 41], where the tuning task can become overwhelming, particularly if



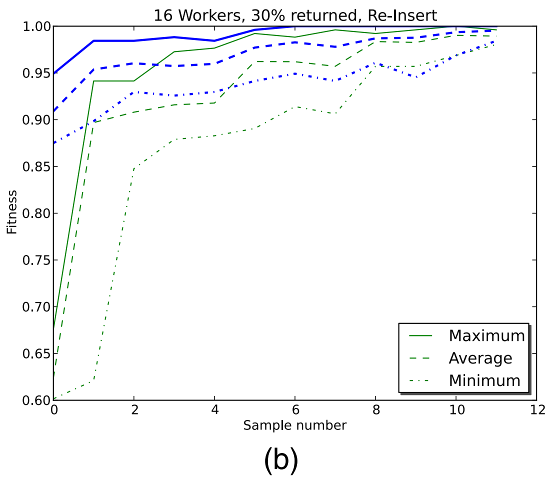


Fig. 9 Evolution of fitness with respect to sample number, using 30 % returned samples and 16 EvoWorkers, for each sample the average fitness was measured at the start (green) and end (blue) of each local evolution, with **a** random algorithm and **b** reinsertion algorithm

the number of islands is large. Therefore, the proposal in [23] is to set the parameter values randomly, without a tuning or self-adaptive process whatsoever. The RPSS approach sets the parameters of each deme randomly at the beginning of the run, a very simple and apparently naive approach. Results suggest that when the number of distributed process is large enough, algorithm parameters can be set randomly and still achieve good results. Therefore the goal of this section is to evaluate RPSS on an EvoSpace Pool-EA.

To gauge the effectiveness of RPSS on a Pool-EA, it is compared with three different parametrization strategies, similar to what is done in [23, 41]. All methods are compared based on average performance over a set of runs. First, the simplest approach consists on setting all of the EvoWorker parameters homogeneously. To do this, 200 random parametrizations are created, setting crossover and mutation probability in the range of [0,1], sample size in the range [12,24] and generations in the range of [5,30]. The average performance of these runs characterizes the random-homogeneous parametrization, denoted Average-Homogeneous. From these runs, the best configuration is chosen, the one that achieved the best results, and then 20 independent runs are carried out on each problem, this method is called Best-Homogeneous. Finally, the random-heterogeneous-parametrization is considered, where the parameters of each EvoWorker are set independently at random at the beginning of each run; 20 independent runs are performed, the method is denoted as Average-Heterogeneous. The algorithms are evaluated using the P-Peaks problems.

Table 4 GA and EvoWorker parameters used in Section 6.1

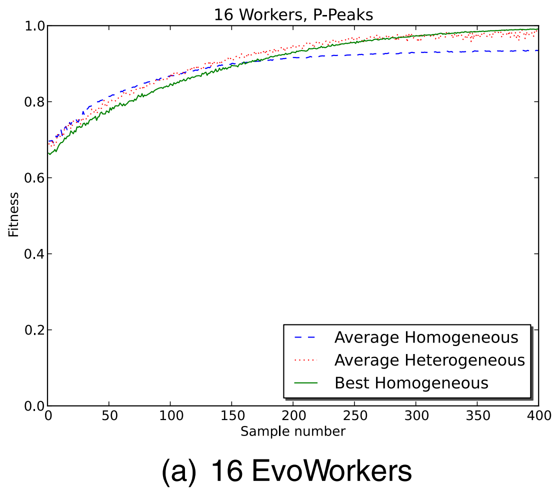
GA Parameters	
Tournament size	4
Crossover rate	0.85
Population Size	512
Mutation probability	0.5
Independent bit flip probability	0.02
EvoWorker Parameters	
Sample Size	16
Generations	128
Other Parameters	
PiCloud Worker Type	Realtime
Number of Workers	4,8,16
Return Sample Probability	30 %,50 %,90 %
Number of Executions	30

Experiments are carried out using a different number of EvoWorkers on each problem. The first group of runs are done with 16 EvoWorkers, and the second with 120. Based on [23, 41], it is assumed that with an increased number of workers the RPSS approach will achieve relatively better results, much closer to the Best-Homogeneous configuration. This is particularly important, since increasing the number of EvoWorkers greatly magnifies the dimensionality of the tuning problem. Results are summarized by tracking how the best solution found so far varies with respect to the total number of samples taken from the EvoStore. These results are presented in Fig. 10, showing the average performance for each of the three methods.

First, with 16 EvoWorkers we can see a clear trend, the Average Heterogeneous configuration is similar with the Best Homogeneous configuration, depicted in Fig. 10a. This is a promising result, since the heterogeneous configuration did not require any parameter tuning, while the best homogeneous configuration is chosen from a set of 200 runs. Moreover, the plots show that using an homogeneous configuration with random values achieves noticeably inferior performance. When the number of EvoWorkers is increased, which is shown in Fig. 10b, a similar trend appears, however the differences among the algorithms is reduced. Future work will compare this technique with other known parametrization methods [7], to determine the best parametrization strategies for a wider range of EAs and problem domains. Nevertheless, these results suggest that a random heterogeneous parametrization following the RPSS approach could be used as a simple off-the-shelf parametrization approach for practitioners interested in using the ESM.

7 Concluding Remarks

This paper presents the EvoSpace model for the development of pool-based EAs, which is designed to exploit computing resources over a network and can be implemented to run directly on the cloud. Pool-EAs present several interesting properties that



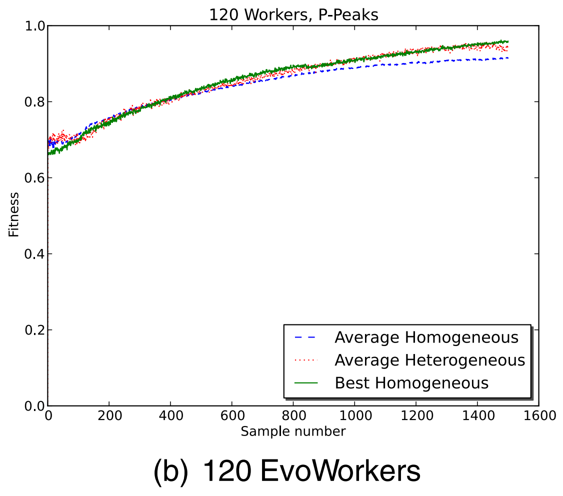


Fig. 10 Convergence plots for the P-Peaks with 16 (**a**) and 120 (**b**) EvoWorkers

are not present in standard EAs, such as the storage of the population in a central store and performing the evolutionary process in an asynchronous and distributed manner through client machines, which in the ESM are referred to as EvoWorkers. In this sense, what is particularly interesting about a Pool-EA is that it incorporates features from natural evolution that are abstracted away in common EA implementations. Moreover, they are intended as means by which evolutionary computation techniques can incorporate modern computing frameworks. However, Pool-EAs also present several noteworthy design challenges, that must be addressed. The proposed ESM is intended as a conceptual framework that addresses some of these challenges and allows for the development of a variety of search techniques that follow this pool-based approach.

At the implementation level, this work also presents an instance of ESM called EvoSpace-py, an open source software tool that simplifies the deployment of Pool-EAs, as multi-threaded processes or on the cloud. The system has been programmed using the Python programming language, the DEAP library and the Redis key-value database.

Experimental results are presented for a pool-based GA, using standard benchmarks and comparing the system with a standard GA search and an IMEA. Results are encouraging in several respects. First, it seems that the ESM can perform a more efficient search that an IMEA, requiring about half as many fitness function evaluations to find a solution on a standard benchmark. Second, performance of the ESM is equivalent or better than standard search, with the added benefit of improved efficiency and better performance on deceptive problems. Third, results illustrate the benefits of adding client EvoWorkers to the evolutionary process. Fourth, several apparent issues with the Pool-EA approach are studied, regarding the lost connection of unreliable workers and the increased size of the algorithm's parameter space. In both cases, experimental results suggest that the ESM can handle both issues robustly, using built-in mechanisms and a simple parametrization strategy.

This paper is the first to comprehensively describe and contextualize the ESM and provides a comprehensive evaluation with respect to standard evolutionary search. Future work will have to explore the limits and comparative benefits of the ESM, in order to define the proper domain of competence of Pool-EAs in general and the ESM in particular. Moreover, the ESM will be leveraged to implement and deploy a complete web-based Pool-EA, that simplifies the use for other researchers in an easy to setup manner.

Acknowledgements Funding provided by CONACYT (Mexico) Project No. 29537 from the Programa de Estimulo a la Innovación, CONACYT Basic Science Research Project No. 178323, DGEST (Mexico) Research Projects No. 5149.13-P, 5414.14-P and TIJ-ING-2012-110, and IRSES project ACoBSEC financed by the European Commission. Additional funding provided by projects P08-TIC-03903 (Andalusian Regional

Government), TIN2011-28627-C04-02 (Spanish Ministry of Science and Innovation), project CANUBE (http://canube.wordpress.com) awarded by the CEI-BioTIC UGR. Regional Government Junta de Extremadura, Consejería de Economía, Comercio e Innovación and FEDER, project GRU10029.

References

1. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. John Wiley & Sons (2005)
2. Allcock, B., Bester, J., Bresnahan, J., Chervenak, A.L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., Tuecke, S.: Data management and transfer in high-performance computational grid environments. Parallel Comput. **28**(5), 749–771 (2002)
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
4. Baxevanidis, K., Davies, H., Foster, I., Gagliardi, F.: Grids and research networks as drivers and enablers of future internet architectures. Comput. Netw. **40**(1), 5–17 (2002)
5. Bollini, A., Piastra, M.: Distributed and persistent evolutionary algorithms: A design pattern. In: Proceedings of the Second European Workshop on Genetic Programming, pp. 173–183. Springer-Verlag, London, UK, UK (1999)
6. Cahon, S., Melab, N., Talbi, E.G.: ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. J. Heuristics **10**(3), 357–380 (2004)
7. Cantú-Paz, E.: Parameter setting in parallel genetic algorithms. In: Lobo, F.G., Lima, C.F., Michalewicz, Z. (eds.) Parameter Setting in Evolutionary Algorithms, volume 54 *Studies in Computational Intelligence*, pp. 259–276. Springer (2007)
8. Cole, N., Desell, T.J., Gonzalez, D.L., de Vega, F.F., Magdon-Ismail, M., Newberg, H.J., Szymanski, B.K., Varela, C.A.: Evolutionary algorithms on volunteer computing platforms: The milkyway@ home project, pp. 63–90. Springer (2010)
9. Cotillon, A., Valencia, P., Jurdak, R.: Android genetic programming framework. Proceedings of the 15th European conference on Genetic Programming, EuroGP'12, pp. 13–24. Springer, Berlin, Heidelberg (2012)
10. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S.: Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing **6**(2), 86–93 (2002)
11. De Jong, K.A., Potter, M.A., Spears, W.M.: Using problem generators to explore the effects of epistasis. In Bäck T. (ed.) Proceedings of the 7th International Conference on Genetic Algorithms, 338–345. Morgan Kauffman (1997)
12. De Jong, K.A., Spears, W.M.: An analysis of the interacting roles of population size and crossover in genetic algorithms. Proceedings of the 1st Workshop on Parallel Problem Solving from Nature, PPSN I, pp. 38–47. Springer, London (1991)
13. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer (2003)
14. Fazenda, P., McDermott, J., O'Reilly, U.M.: A library to run evolutionary algorithms in the cloud using mapreduce. In: di Chio, C., et al. (eds.) Applications of Evolutionary Computation, volume 7248 *LNCS*, pp. 416–425. Springer, Berlin Heidelberg (2012)
15. Fernández De Vega, F., Olague, G., Trujillo, L., Lombraña González, D.: Customizable Execution Environments for Evolutionary Computation Using BOINC + Virtualization. Nat. Comput. **12**(2), 163–177 (2013)
16. Fortin, F.A., Rainville, F.M.D., Gardner, M.A., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. J. Mach. Learn. Res. **13**, 2171–2175 (2012)
17. Foster, I., Kesselman, C. (eds.): The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers Inc., San Francisco (1999)
18. Garcia-Arenas, M., Merelo, J.J., Mora, A.M., Castillo, P., Romero, G., Laredo, J.: Assessing speed-ups in commodity cloud storage services for distributed evolutionary algorithms. In: 2011 IEEE Congress on Evolutionary Computation (CEC), pp. 304–311. IEEE (2011)
19. Garcia-Valdez, M., Mancilla, A., Trujillo, L., Merelo, J.J., Fernandez-de Vega, F.: Is there a free lunch for cloud-based evolutionary algorithms? In: 2013 IEEE Congress on Evolutionary Computation (CEC), pp. 1255–1262 (2013)
20. Garcia-Valdez, M., Trujillo, L., Fernández de Vega, F., Merelo Guervós, J., Olague, G.: Evospace-interactive: A framework to develop distributed collaborative-interactive evolutionary algorithms for artistic design. In: Machado, P., et al. (eds.) Evolutionary and Biologically Inspired Music, Sound, Art and Design, LNCS, vol. 7834, pp. 121–130. Springer, Berlin Heidelberg (2013)
21. García-Valdez, M., Trujillo, L., Fernández de Vega, F., Merelo Guervós, J.J., Olague, G.: EvoSpace: A Distributed Evolutionary Platform Based on the Tuple Space Model. In: Esparcia-Alcázar, A., et al. (eds.) Applications of Evolutionary Computation, LNCS, vol. 7835, pp. 499–508. Springer, Berlin Heidelberg (2013)
22. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)
23. Gong, Y., Fukunaga, A.: Distributed island-model genetic algorithms using heterogeneous parameter settings. In: IEEE Congress on Evolutionary Computation, pp. 820–827. IEEE (2011)
24. Klein, J., Spector, L.: Unwitting distributed genetic programming via asynchronous JavaScript and XML. Proceedings of the 9th annual conference on Genetic and evolutionary computation, GECCO '07, pp. 1628–1635. ACM, New York (2007)
25. Kramer, O.: Self-Adaptive Heuristics for Evolutionary Computation, Studies in Computational Intelligence, vol. 147. Springer (2008)
26. Langdon, W.B. In: Keijzer, M., O'Reilly, U.M., Lucas, S.M., Costa, E., Soule, T. (eds.): Global distributed evolution of l-systems fractals, pp. 349–358. Springer (2004)
27. Lobo, F.G., Lima, C.F., Michalewicz, Z.: Parameter Setting in Evolutionary Algorithms. Springer Publishing Company, Incorporated (2007)
28. Merelo, J.J., Castillo, P., Mora, A., Esparcia-Alcázar, A., Rivas-Santos, V.: NodEO, a multi-paradigm distributed

evolutionary algorithm platform in javascript. Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion, pp. 1155–1162. ACM (2014)
29. Merelo, J.J., Fernandes, C.M., Mora, A.M., Esparcia, A.I.: Sofea: A pool-based framework for evolutionary algorithms using couchdb. Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, GECCO '12, pp. 109–116. ACM, New York (2012)
30. Merelo, J.J., Mora, A., Fernandes, C., Esparcia-Alcazar, A., Laredo, J.: Pool vs. island based evolutionary algorithms: An initial exploration. In: P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on, pp. 19–24 (2012)
31. Merelo-Guervós, J.J., Mora, A., Cruz, J.A., Esparcia, A.I.: Pool-based distributed evolutionary algorithms using an object database. Proceedings of the 2012 European conference on Applications of Evolutionary Computation, EvoApplications'12, pp. 446–455. Springer, Berlin, Heidelberg (2012)
32. Merelo-Guervos, J.J., Mora, A., Cruz, J.A., Esparcia-Alcazar, A.I., Cotta, C.: Scaling in distributed evolutionary algorithms with persistent population. 2012 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. IEEE Comuter Society (2012)
33. Merelo Guervos, J.J., Valdivieso, P.A.C., Laredo, J.L.J., García, A.M., Prieto, A.: Asynchronous distributed genetic algorithms with JavaScript and JSON. IEEE Congress on Evolutionary Computation, pp. 1372–1379. IEEE (2008)
34. Oram, A. (ed.): Peer-to-Peer: Harnessing the Power of Disruptive Technologies. O'Reilly & Associates, Inc., Sebastopol (2001)
35. Paechter, B., Back, T., Schoenauer, M., Sebag, M., Eiben, A., Merelo, J.J., Fogarty, T.: A distributed resource evolutionary algorithm machine (DREAM). In: Evolutionary Computation, 2000. Proceedings of the 2000 Congress on, vol. 2, pp. 951–958 vol.2 (2000)
36. Roy, G., Lee, H., Welch, J.L., Zhao, Y., Pandey, V., Thurston, D.: A distributed pool architecture for genetic algorithms. Proceedings of the Eleventh conference on Congress on Evolutionary Computation, CEC'09, pp. 1177–1184. IEEE Press, Piscataway, NJ, USA (2009)
37. Schmidt, M., Lipson, H.: Distilling free-form natural laws from experimental data. Science **324**, 81–85 (2009)
38. Secretan, J., Beato, N., D'Ambrosio, D.B., Rodriguez, A., Campbell, A., Folsom-Kovarik, J.T., Stanley, K.O.: Picbreeder: A case study in collaborative evolutionary exploration of design space. Evol. Comput. **19**(3), 373–403 (2011)
39. Sherry, D., Veeramachaneni, K., McDermott, J., O'Reilly, U.M.: Flex-gp: Genetic programming on the cloud. In: di Chio, C., et al. (eds.) Applications of Evolutionary Computation, LNCS, vol. 7248, pp. 477–486. Springer, Berlin Heidelberg (2012)
40. Talukdar, S., Baerentzen, L., Gove, A., De Souza, P.: Asynchronous teams: Cooperation schemes for autonomous agents. J. Heuristics **4**(4), 295–321 (1998)
41. Tanabe, R., Fukunaga, A.: Evaluation of a randomized parameter setting strategy for island-model evolutionary algorithms. IEEE Congress on Evolutionary Computation, pp. 1263–1270. IEEE (2013)
42. Thierens, D.: Scalability problems of simple genetic algorithms. Evol. Comput. **7**, 331–352 (1999)
43. Trujillo, L., Valdez, M.G., de Vega, F.F., Merelo-Guervós, J.J.: Fireworks: Evolutionary art project based on EvoSpace-interactive. IEEE Congress on Evolutionary Computation, pp. 2871–2878. IEEE (2013)
44. Varia, J.: Cloud architectures. White Paper of Amazon (2008)
45. Vecchiola, C., Kirley, M., Buyya, R.: Multi-objective problem solving with offspring on enterprise clouds. CoRR abs/0903.1386 (2009)
46. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. J Internet Serv Appl **1**(1), 7–18 (2010)