Noname manuscript No.
(will be inserted by the editor)

First Author · Second Author

the date of receipt and acceptance should be inserted later

Abstract **Keywords** First keyword · Second keyword · More

1 Introduction

Over the last decade, the landscape in computing technology has undergone a noticeable shift, moving away from isolated, serial and synchronous systems, towards distributed, ubiquitous and shared computing resources, exploiting network infrastructure and what is known as the Cloud computing model. This shift is not easy to make, it requires a dedicated and coordinated effort to port systems from one paradigm to another. This paper is part of an evergrowing effort in the field of evolutionary algorithms (EAs) to develop new software tools that exploit the nuances of distributed and asynchronous computational systems. In particular, this works centers around EvoSpace, a pool-based EA (PEA) that presents unique features and allows for non-traditional dynamics during an evolutionary search .

EvoSpace is built around a central repository or population store, where the population of an EA is kept following a tuple-space model . Distributed clients, or workers, connect with the central store and take a subset of individuals over the network, on which they perform a partial evolutionary process, and then return the newly evolved subpopulation to the central store where the new individuals are reinserted, at which point they can be taken by other clients. Workers, besides being distributed across a network or over the Cloud, 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 one presented with EvoSpace, allows for the design of EAs that include algo-

F. Author
first address
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: fauthor@example.com

S. Author
second address

rithmic features that are present in natural evolutionary systems, but are difficult to reproduce in sequential and synchronous processes .

This paper presents a detailed description of EvoSpace, building upon previous contributions made by the authors , thus providing a comprehensive analysis of the system at both the conceptual and implementation levels, discussing how the system can be employed for standard search and optimization. Furthermore, several design issues are addressed, that make EvoSpace, as well as other PEAs, unique among the current EA landscape. Firstly, distributed systems have to account for unreliable clients, and how lost connections can compromise system performance, an issue that is experimentally evaluated in this paper. Secondly, by performing the evolutionary process over a number of distributed workers, the system's parameter space grows exponentially, thus greatly enhancing the undesirable feature of most EAs, that of tuning and parameterizing the algorithm for a new problem. This issue is addressed by using a simple, but efficient, parametrization method originally developed for Island model EAs .

The remainder of the paper proceeds as follows. First, a comprehensive discussion pertaining to related work is presented in Section 2. Afterwards, the EvoSpace system is detailed in Section 3, discussing system performance on some standard genetic algorithm (GA) benchmarks, and summarizing how EvoSpace can be leveraged for interactive applications. Section 4 then deals with what might be considered as possible shortcomings of the EvoSpace approach, and proposes useful solutions to these issues which are validated experimentally. Finally, a summary of this work, conclusions and future work is given in Section 5.

2 Related Work

Exploiting Internet resources in the development of new EAs has received substantial attention. Probably the most direct way to accomplish this is to run an EA over a web browser. For instance, Klein and Spector and Merelo et al. developed EAs using Javascript, distributing fitness function evaluations over the web, 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 in the work of Cotillon et al. , where they extend the concept of browser-based EAs on to the Android OS that can be executed on a large number of mobile devices. However, using the browser to run an EA comes at a cost, given that code runs far less efficiently. Therefore, probably the most successful examples of web-based EAs come from the area of interactive evolution, where users can visit a web page, browse candidate individuals and participate in the evolutionary process by evaluating individuals. An early example is presented by Langdon , were users evaluate evolving agents expressed as fractal structures. Such a strategy can also be found in the more recent works of Secretan et al. and Clune and Lipson , who respectively developed web pages for the interactive evolution of 2D images and 3D sculptures using a neuro-evolution algorithm. Indeed, a variant of EvoSpace, called EvoSpace-i, has also been developed explicitly for these type of interactive evolution of artistic artifacts .

Other researchers have exploited other network based technologies to distribute an EA. For instance, Cole et al. use the popular Berkeley Open Infrastructure for Network Computing to distribute an evolutionary algorithm, 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. , who also distributes multiple EA runs using a volunteer computing network through BOINC. Garcia-Arenas et al. used the popular file sharing service Dropbox as a storage server for an EA. Jiménez-Laredo et al. uses a peer-to-peer architecture to distribute a genetic programming (GP) algorithm over a network. Similarly, the DREAM framework presented in also distributes an EA using a peer-to-peer 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. using distributed model based on the Farmer/Worker paradigm. Recently, researchers have also begun to integrate EAs fully into the Cloud computing model. For instance, the Eureqa [1] GP-based modeling tool originally developed by Schmidt and Lipson , marketed by nutonian, integrates easily with Amazons EC2 Cloud service for faster computation. Integrating EAs more fully into the Cloud model, two noteworthy examples are Offspring framework by Vecchiola et al. and the FlexGP system developed by Sherry et al. . 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 .

All of these works show great promise in extending EAs towards modern computing models. However, the current work is more closely related to what can be referred to as pool-based EAs or PEAs. Such approaches are related to more general systems such as A-Teams , a problem-solving multi-agent architecture based on a strongly cyclic network. G. Roy et al. 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 database. Bollini and Piastra developed a system that decouples population storage from the evolutionary operations.

The most recent work that is comparable with EvoSpace is the SofEA algorithm proposed by Merelo et al. **??**. SofEA is an evolutionary algorithm mapped to a central CouchDB object store. It 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 have been executed concurrently in the reported experiments. 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

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

maintains a fine-grained control of each individual within the population that limits the ability of each to interact with remote clients. 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. Furthermore, if an individual *dies* (or is eliminated), then another copy of it cannot be reintroduced into the population, even if might be useful at a later stage of the search. Finally, by using the individual chromosome as ID, it appears problematic to extend SofEAs implementation to representations such as that of tree based GP, an extension that is easily accomplished with EvoSpace.

3 The EvoSpace System

Computational resources are offered as different types of services within the Cloud computing model **?**. For example, infrastructure as a service (IaaS) such as Amazon's EC2, or Platform as a Service (PaaS) such as Google's App Engine. However, within Evolutionary Computation (EC), most algorithms have not been ported to the Cloud computing model.

EvoSpace works as a population store for the development of EAs that are intended to run on the Cloud. Within the Cloud computing model, EvoSpace is conceived as a Platform as a Service (PaaS) component, where users can create populations of individuals on demand without the need to install software or invest in additional hardware. EvoSpace is designed to be versatile, since the system that manages the population is decoupled from any particular EA. Client processes, called EvoWorkers, dynamically and asynchronously interact with the EvoSpace store and perform the desired evolutionary processes. EvoWorkers can reside on remote clients or on the platform server if need be. Software as a Service (SaaS) applications could also be developed using EvoSpace, where users could run entire experiments on the Cloud or implement interactive applications that are completely hosted by the service . EvoSpace is well suited for this kind of environment, since it can be accessed by any web enabled device and is designed to be robust to lost connections. Figure 1 presents a conceptual diagram of how EvoSpace fits within the Cloud computing model.

3.1 System Design

Basically, EvoSpace consists of two main components. First, the EvoSpace container object, which resides on a central server, based on a tuplespace representation, that stores the evolving population. The second component consists of the remote EvoWorkers, which are distributed process that are executed on client machines that connect to the server. EvoWorkers execute the actual evolutionary process, while EvoSpace only serves as a population repository. In summary, EvoWorkers extract a small subset of the population, and use it as the initial population for for a short evolutionary algorithm that is executed locally on the client machine. Afterwards, the evolved population from each EvoWorker is returned and reinserted into the EvoSpace container. The server-side Respawn process is used to alleviate

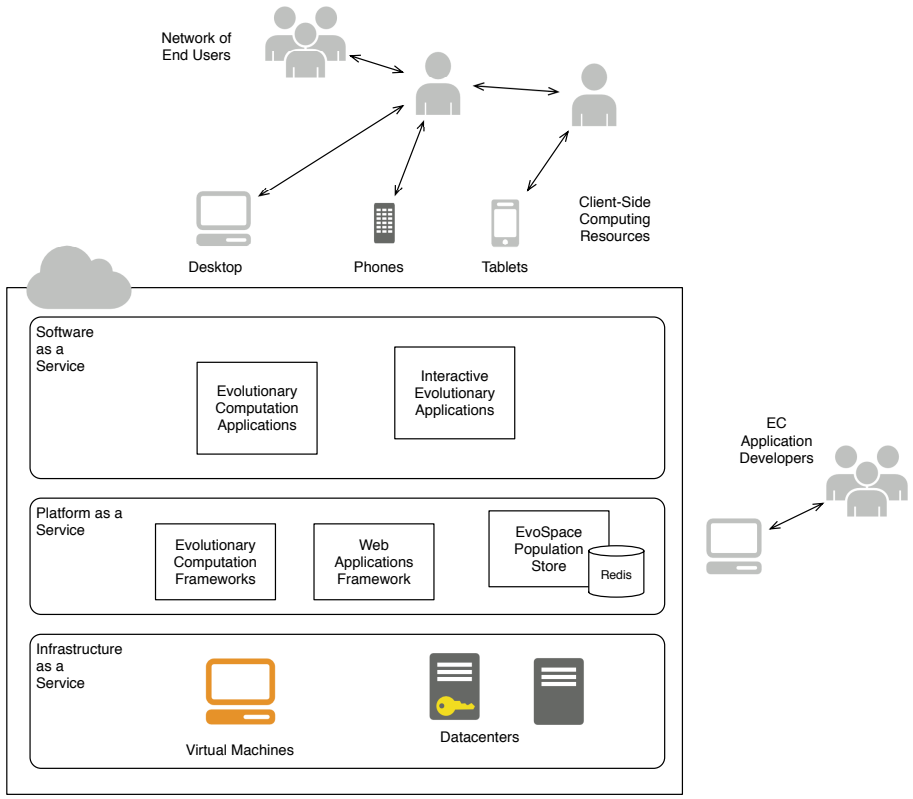


Fig. 1 Conceptual diagram of the Cloud Computing Model and of how EvoSpace can fit within, using it as a PaaS or possible using it to develop a SaaS.

possible problems that might occur during the search, such when the EvoSpace container is emptied or when connections are lost with connected EvoWorker clients. Figure 2 presents a graphical illustration of the main components and dataflow within EvoSpace.

3.1.1 The EvoSpace container

EvoSpace is based on the tuple space model, an associatively addressed memory space shared by several processes. A tuple space 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.

EvoSpace is composed by a set of objects ES and a set of interface methods provided by a central server. For an EA system, objects correspond to individuals and all their corresponding features and related information. Objects can be withdrawn, processed and replaced into ES using a specified set of methods. However, EvoSpace is different from other tuple space implementations in the sense that retrieving and reading objects from ES are random operations. Individual objects are

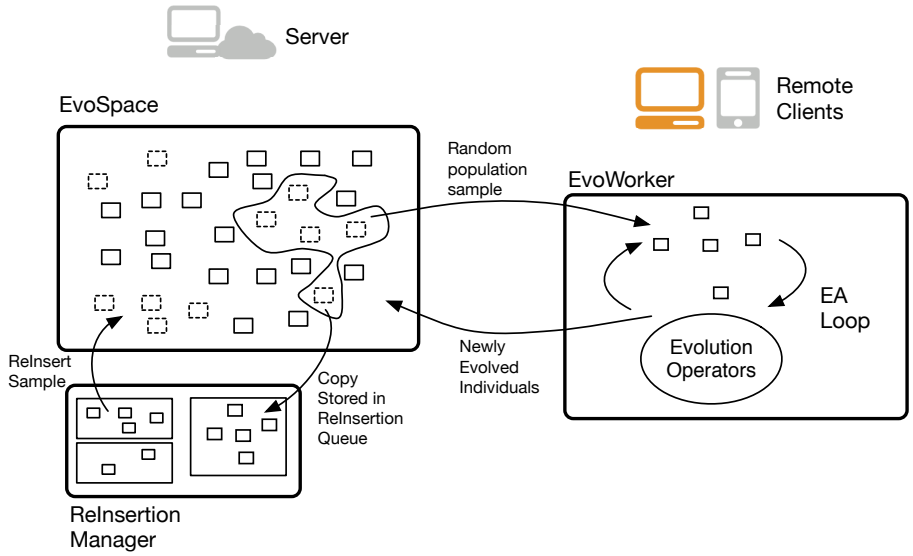


Fig. 2 Main components and dataflow within EvoSpace.

not of high interest when accessing ES. Therefore, EvoSpace offers the following interface methods.

- **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.
- **Take(n):** Returns a random set A, following the similar logic used for $Read()$. However, in this case the sequence of $Take()$ operations provide a temporal dimension to the dynamics of set ES. We can define ES_i as the set at the moment of the $i-th$ $Take()$ operation and A_i as the output. The contents of EvoSapce are then given by $ES_{i+1} = ES_i \setminus A_i$; i.e., the objets taken are effectively removed from ES. The objects taken are also copied to a new set S_i of *sampled objects* 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 $i-th$ $Take()$ operation, such that the contents of EvoSpace are now $ES \cup S_i$ if $S_i \in \mathcal{S}$ and ES is left unchanged otherwise.
- **Insert(A):** This method represents the union operation $ES \cup A$.
- **Replace(A,i):** Similar to $Add()$, 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 $ES \cup (A \cap ES)$.

Algorithm 1 The server-side **EvoSpaceServer** process.

```
EvoSpace ← new EvoSpace
EvoSpace.active ← true
while EvoSpace.active do
    read method
    return EvoSpace.method()
end while
```

3.1.2 Individuals

As stated above, objects in ES represent individuals in 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. In this case, keys represent specific properties of each object and the values can be of different types, such as numbers, strings, lists, tuples or other dictionaries. In the current implementation, the objects in ES 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. The **fitness** dictionary for each individual; allowing the system to store multiple fitness values for multi-objective or dynamic scenarios. A **parents** dictionary with identifiers of the individual(s) from which it was produced. Finally, a **GeneticOperator** string that specifies the operator that produced it.

3.1.3 The EvoSpace Server Processes

The EvoSpace architecture employs a client-server architecture with a shared memory container. On the server side, a process called **EvoSpaceServer** is executed, which creates and activates a new EvoSpace container object and waits for requests to execute interface methods; see Algorithm 3.1.3. Additionally, on the server three more processes are executed, these are: **InitializePopulation**, **ReInsertionManager** and **EvolutionManager**; see algorithms 3.1.3, 3.1.3, and 3.1.3. **InitializePopulation** is executed once, its goal is obvious, initialize the population by adding $popsize$ random individuals. The function that creates new individuals depends on the problem and representation used. **ReInsertionManager** is used as a failsafe process that periodically checks (every wt seconds) if the size of the population in ES falls below a certain threshold or if the time after the last $ReInsert()$ is greater than $next_r$. If any of these conditions are satisfied, then rn subsets $S_i \in \mathcal{S}$ are reinserted into ES using the $ReInsert()$ method. Moreover, the **ReInsertionManager** makes a $ReInsert(i)$ call when $EvoWorker_i$ (see below) looses a connection with the server, thus recovering the population sample. Finally, **EvolutionManager** periodically checks if a termination condition is satisfied, which is checked by the $isOver()$ method. This method can be implemented according to the needs of the evolutionary search. For instance, $isOver()$ can check if an optimal solution is found or if a maximum number of function evaluations have been performed.

3.1.4 The EvoSpace Clients: EvoWorkers

The second component of the proposed model are the processes executed on client machines, referred to as **EvoWorkers**, see Algorithm 3.1.4. Each client loads a pro-

Algorithm 2 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}
    $EvoSpace.ES.Add(i)$
    j++
  end for

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

Require: $ri \leftarrow$ respawn interval
Require: $min \leftarrow$ population size threshold
Require: $rn \leftarrow$ number of samples to respawn
Require: $EvoSpace \leftarrow$ Reference to an Evospace Server
Require: $wt \leftarrow$ wait interval
  $next_r \leftarrow +ri$
  while $EvoSpace.active$ **do**
    if $|EvoSpace.ES| \leq min$ OR $currentTime \geq next_r$ **then**
      $EvoSpace.RespawnOld(rn)$
      $next_r \leftarrow currentTime + ri$
    end if
    $wait(wt)$
  end while

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

Require: $EvoSpace \leftarrow$ Reference to an Evospace Server
Require: $t \leftarrow$ time interval
  $terminate \leftarrow$ false
  while NOT terminate **do**
    $wait(t)$
    $terminate \leftarrow isOver(EvoSpace)$
  end while

cess that executes the main code of the EA. The **EvoWorker** process is straightforward, it requests a set of objects A_i from the ES container. Afterwards, the $Evolve()$ function is called where the actual evolutionary cycle is performed. In this scenario, A_i can be seen as a local population on which evolution is carried out for g generations. The result of this evolution is then returned and reinserted into ES, then the EvoWorker can request a new set from ES 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 not taken in the present work.

3.1.5 Implementation

Figure 3 shows a schematic diagram of the proposed architecture and the corresponding information flow with implementation details. Individuals are stored in-memory, using the Redis key-value database. Redis was chosen over a SQL-based management system, or other non-SQL alternatives, because it provides a hash based implementation of sets and queues which are natural data structures for the EvoSpace model. For example, selecting a random key from a set has a complex-

Algorithm 5 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

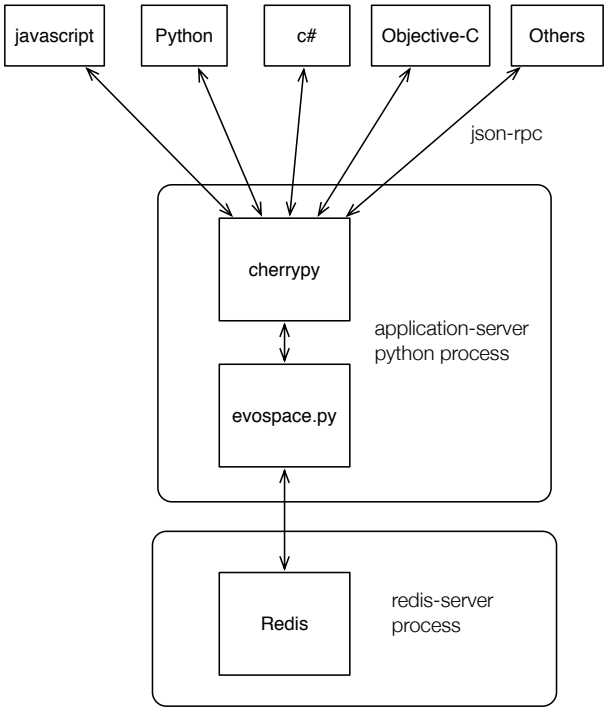


Fig. 3 Evospace Implementation architechture

ity of O(1). The logic of EvoSpace is implemented as a Python module exposed as a Web Service using Cherrypy and django http frameworks. The EvoSpace web service can interact with any language supporting json-rpc or ajax requests. The EvoSpace modules and workers in JavaScript, JQuery and python are available with a Simplified BSD License from `http://github.com/mariosky/EvoSpace`.

As EvoSpace is only the population store, EvoWorkers must implement the genetic operators of a particular EA. As stated earlier, our objective is to let researchers use common tools as in their local setting. Given that EvoSpace is implemented as a Python module, a simple way to implement the EvoWorkers is to use

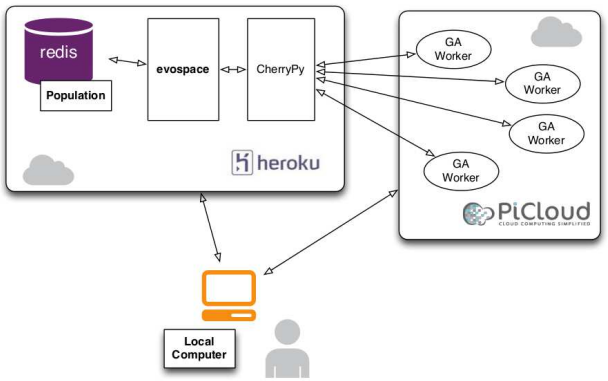


Fig. 4 EvoSpace cloud-based architecture.

the DEAP (Distributed Evolutionary Algorithms in Python) library **?**, for the basic non-distributed GA library. Three methods were added to the local algorithm: getSample() and putBack(); and another for the initialization of the population using DEAPs own methods. For instance, to generate the initial population, a local initialize() is called and the population sent to EvoSpace.

3.2 EvoSpace on the Cloud

This section describes how EvoSpace can be configured to run on the Cloud using Heroku and PiCloud; a schematic view of the cloud architecture is shown in figure 4.

3.3 Evospace as a Heroku Application

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 (including systems used for database and queuing, for instance). These process types are the prototypes from which one or more dynos can be instantiated; if the number of requests to the server increases 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 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; in our case and in the more demanding configurations of our experiments, the web process was scaled to 20 dynos. Instructions and code for deployment is available at `http://www.evospace.org/software.html`

3.4 Evoworkers as PiCloud Jobs

PiCloud is a Platform as a Service (PaaS), with deep Python integration; using a library, Python 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 core available, the job is assigned to it. Both Heroku and PiCloud platforms are deployed on top of Amazon Web Services (AWS) infrastructure in the US-EAST Region. This ensures minimal latency and a high bandwidth communication between the services, and there is no charge for data transfer costs between both services. For the experiments type c1 and c2 Real Time workers where used. The code for the EvoWorkers implementation and experiment data is publicly available from a github repository `https://github.com/mariosky/evoPar2014`.

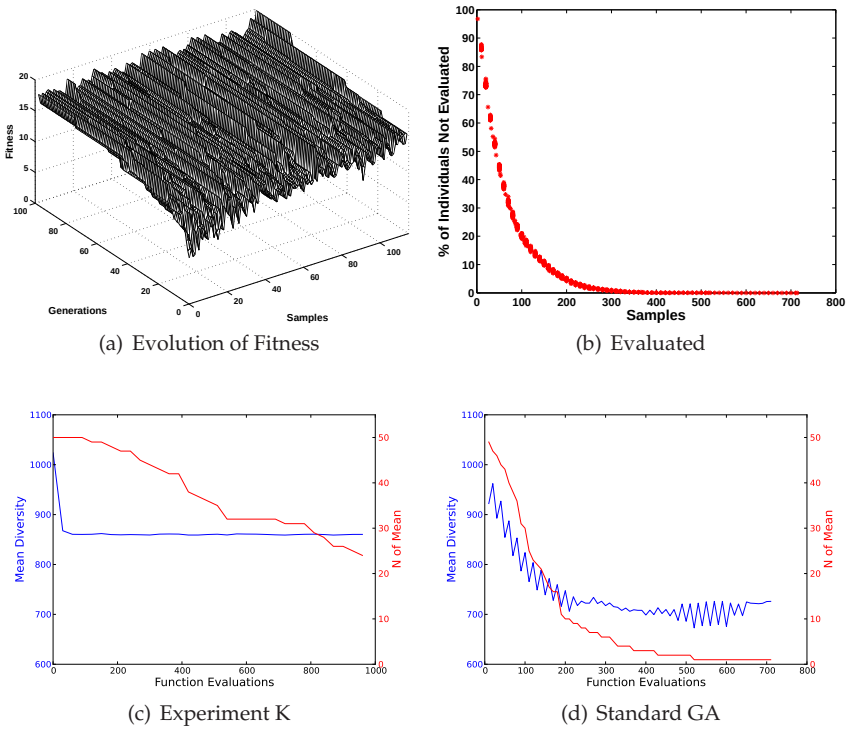
3.5 Experiments and Results

This section presents an experimental evaluation of EvoSpace using two well-known benchmarks for GAs; the K-trap problem and the P-Peaks function .

3.5.1 Experiment A: K-trap Function

This problem presents a multi-modal and deceptive fitness landscape. Table 1 summarizes the different experimental configurations tested in this work, based on the K value, number of EvoWorkers, the sample size taken by each worker and the chromosome length used in each K-trap problem. The number of individuals in the EvoSpace repository is set to 1024 for 4-trap experiments and to 4096 for 5-trap. Moreover, Table 2 summarizes the shared parameters and algorithmic configuration. The maximum number of samples that each EvoWorker can take from EvoSpace is 1000. For comparison, a standard GA is applied to each benchmark problem. For the 4-trap problem, the maximum number of generations is 4000, for the 5-trap problem it is 1000, this bounds the number of function evaluations based on the maximum number of samples taken from all of the EvoSpace runs. In total, 50 runs were performed of 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 1; 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 10, we can see that the evolution on the last sample taken from EvoSpace reaches the global optimum. The overall performance of EvoSpace is summarized in Table 3, which shows the total number of runs, out of 50, that found the global optima. EvoSpace outperforms the standard GA on both functions, with a substantial increase in the number of optima found.



(a) Evolution of Fitness (b) Evaluated

(c) Experiment K (d) Standard GA

Fig. 5 (a) Evolution of fitness for a run of experiment K; 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 from experiment K. (c) Evolution of diversity for the basic GA on the 5-trap problem. (d) Evolution of diversity in EvoSpace for experiment K. Plots c-d are double y-axis plots, showing the mean diversity over all runs (dark line) and the number of runs N used to compute the mean.

Table 1 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 length	40	40	40	40	40	40	40	40	40	50	50	50	50	50	50

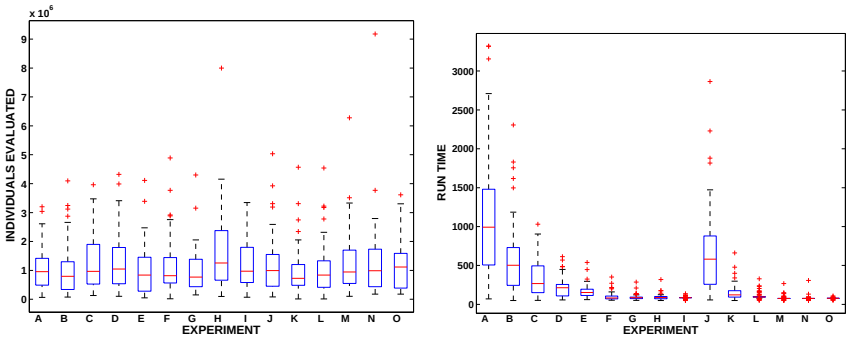
Table 2 Parameters and algorithm configurations for all experiments.

Parameter	Maximum Function Evaluations	Crossover (Prob.)	Mutation (Prob.)	Generations per EvoWorker
Value	100 Gens per worker	Single point (1)	Point (0.06)	100

Table 3 Different experimental configurations used to test the performance of EvoSpace. GA-K are the baseline GA results for the 4-trap and 5-trap functions respectively.

Experiment	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	GA-4	GA-5
Optima found	48	50	50	49	49	50	48	50	50	50	50	50	50	50	50	34	29

Since every EvoWorker takes a random sample of individuals, one concern might be that some individuals in the initial population might not be chosen and



(a) (b)

Fig. 6 (a) Number of evaluated individuals. (b) Total run-time.

evaluated, and valuable genetic material wasted. However, Figure 5b shows that the percentage of individuals within EvoSpace that have not been evaluated quickly decreases as the initial samples are taken. Another interesting question is how the EvoSpace search impacts diversity, given by the sum of the pairwise Hamming distances of all individuals within the population **?**. Figure 5c shows how diversity evolves for experiment K, and Figure 5d shows the same for the basic GA[2] These plots show how the standard GA has a problem maintaining diversity, which leads to its poor performance shown in Table 3. On the other hand, EvoSpace maintains a more diverse population which leads to a better exploration of the search space and more successful runs.

Finally, a comparison of the amount of computational effort required in each experiment is given in Figure 8, which shows boxplots of all runs in each experiment. Figure 8a plots the total number of individuals evaluated in each run, which is similar in all experiments; while Figure 8b compares the total run time in seconds. The figures show that run time is reduced significantly as the number of EvoWorkers increases, as expected.

3.5.2 Experiment B: P-Peaks

The P-Peaks problem was chosen because the problem and the computing resources needed for the search can be appropriately scaled. Proposed by De Jong et al. in **?**, a gneralization of the version in **?**, 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 $\mathbf{x}$ 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

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

[2] The time scale is given by the number of evaluated individuals, in increments that correspond to the number of individuals evaluated in 10 samples from experiment I.

Table 4 GA and EvoWorker parameters for experiments.

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	Realtime, Standard
Number of Workers	2,4,8,16,28
Number of Executions	30

Table 5 Average Times and Evaluations of 30 executions in a local computer.

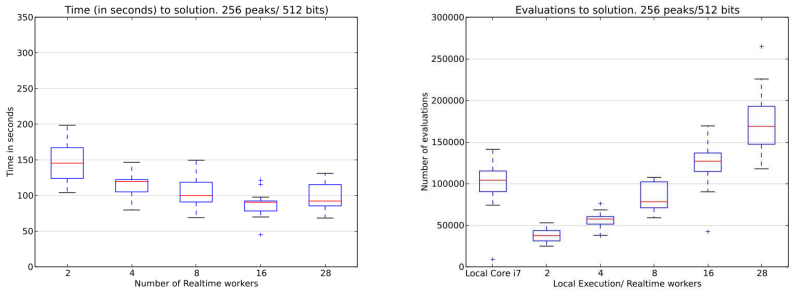
Average time in seconds	Average number of evaluations
1567.36	100690.5

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 4.

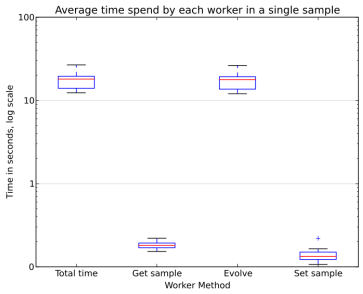
As a baseline execution, the experiment was repeated in a local computer. The specifications for the local computer are as follows, a 2.2 Ghz Intel Core i7 processor, 16 GB of 1333 DDR3 memory, and Mac OS X 10.7.5 operating system. All the software was executed with a Python interpreter version 2.7.2 for 64-bit architectures.

The problem required considerable computational time: on the local computer each run took an average of 1567.36 seconds to find the optimal solution (see table 5). The execution used a single core in the computer 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 workers were used, clocking at less than 180s even in the worst cases.

Average times for the configuration, using Realtime cores in PiCloud, are presented in Figure 7(a). It can be seen that incrementing the number of workers reduced the time to solution, but only up to 16 workers. With 28 workers time to solution did not improve. This is related to the increase in the number of evaluations needed to find the optima, which is shown in Figure 7(b). This figure shows that the number of evaluations needed to find the solution increases as more workers are used. This behavior has an impact on the time to solution, because each evaluation (as stated earlier), is computationally expensive. From 2 to 8 workers the number of evaluations remains less than in the baseline GA, but from that point



(a) Time to solution (b) Number of Evaluations



(c) Worker Costs

Fig. 7 (a) Time required to solution. (b) Number of evaluations to solution. (c) Time of worker's methods, GetSample(), Evolve(), PutSample().

on the number is higher. A possible reason for this is that as the number of workers increases, the number of individuals that remain in the population waiting to be replaced decreases. As there are fewer individuals in the population the probability of taking the same individuals when replacing a sample is increased. This problem is also reported by Merelo et al. in **?**, where they present a PEA architecture with dissimilar workers.

For other problems where the evaluation of individuals is not very demanding, the added cost of communication between EvoSpace and EvoWorkers can become a concern. In these experiments this cost is negligible, Figure 7(c) shows boxplots of the time required two perform the three main methods on the EvoWorkers from 30 runs of the 28 Realtime workers experiment.

3.6 Discussion

4 Overcoming Problems and Limitations

The previous section showed how EvoSpace can be used to implement a distributed and asynchronous EA, with strong results on two benchmark tests. However, there are possible downsides to using a PEA such as EvoSpace, such as a low bandwith, abandoned work, system security and privacy, all of which are important issues.

However, in what follows we focus on two critical issues, lost connections due to the unreliability of connected workers, and algorithm parametrization, a common issue with almost all EAs that is severely amplified in a PEA approach.

4.1 Unreliable Workers

In this section, the effect of node unavailability in an EvoSpace EA is assessed. The EvoSpace model contrasts with the use of a global queue of tasks and implementations of map-reduce algorithms, such as in , with several benefits relevant to concurrency control and workload distribution. For instance, leaving a copy of the individual in the population server free to be pulled by other EvoWorkers will result in redundant work and this could be costly if the task at hand is time consuming. Moreover, 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, EvoSpace uses a simple 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.

This section evaluates the effect that 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. 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, with has the added effect of increasing diversity within the population.

The algorithm stops when reaching the optimum value, or when all workers pulled 100 samples. To simulate unreliable workers each worker 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%. Experiments where carried out for 4, 8 and 16 EvoWorkers. Although supported by EvoSpace, time outs were not chosen as triggers to feed the population with new individuals, the population size was used instead. The population size is a better threshold as it is more critical to GA performance, since when 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 population; the number of individuals fed to the population was 128, or 8 samples when the reinsertion algorithm was used.

Figure 8(a) 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 workers, shown in Figure 8(b), there was a marginal decrease in overall time; and results where similar to those found in the experiments with 4 workers. Figure 8(c) shows results for 16 workers, when there was only a 30% chance of returning a sample the rate of reinsertion was high, approximately once every 35 samples. In this case, the insertion of random individuals resulted

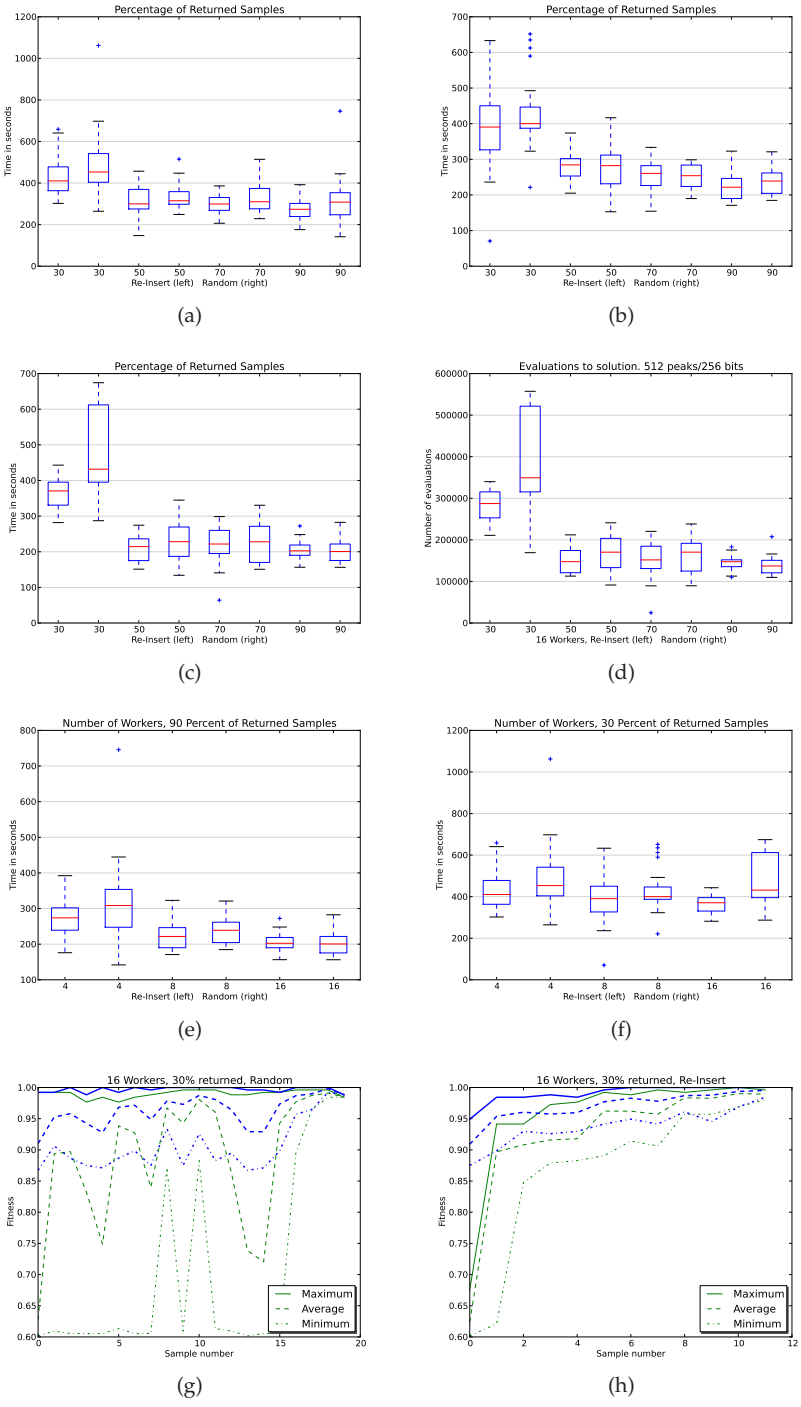


Fig. 8 Unreliable workers: (a) Time to solution, 4 Workers; (b) Time to solution, 8 Worker; (c) Time to solution, 16 Workers; (d) Number of evaluations, 16 Workers; (e) Time to solution, 90% returned samples; (f) Time to solution, 30% returned samples; (g-h) Fitness by sample number, 30% returned samples, 16 Workers, for each sample the average fitness was measured at the start (green) and at the end (blue) of the local evolution, with (g) random algorithm and (h) reinsertion algorithm.

in a higher time to solution. For these experiments when the rate of reinsertion is low, both alternatives have similar results, but the reinsertion algorithm is better for situations when starvation is common. It appears that the insertion of random individuals is not detrimental when there are other evolved individuals in the pool. But when the remaining pool almost consists of random individuals, samples pulled by EvoWorkers need to start the search from scratch. If a small number of samples are returned to the pool, the work needed to reach the optimum is increased. Figure 8(d) also shows the number of evaluations needed to reach an optimum for 16 workers. Figures 8(e) and 8(f) show the time required to solution for 30% and 90% of returned samples. For 90% both algorithms had similar speedups when the number of workers is higher. For 30% there was practically no speedup at all. The reinsert algorithm, although not significant, had a consistent decrease in time.

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 8(g) shows the evolution of fitness with the random insertion algorithm, where as expected initial fitness drops at certain points, when random insertion occurs, while average final fitness is also compromised. Figure 8(h) shows results for the reinsertion algorithm, with more characteristic convergence curves without substantial fitness drops.

4.2 Parametrization

In general, EAs are sometimes criticized by the large number of parameters they posses, that for real world problems need to be tuned empirically or require additional heuristic processes to be included into the search to adjust the parameters automatically **??**. In the case of a PEAs, 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 **?**.

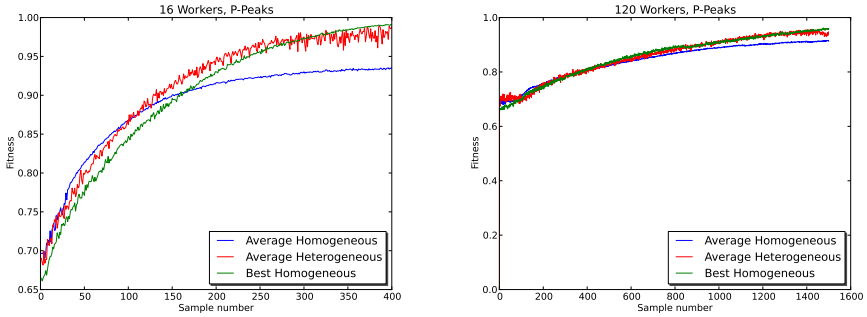
A noteworthy contribution is made by Cantú Paz, who addresses the problem of deriving theoretical models of the effects of parameters related to population size and migration in Island-Model EAs (IMEA) **?**. However, it does not cover the effects of all possible parameters, or the intricacies of a PEA algorithm. Here, we would stress some important differences between PEAs and IMEAs. First, an IMEA presents a fixed topological structure, with a predefined interaction protocol among each evolving population, this leads to a coordinated, or even synchronized, interaction between the islands. On the other hand, a PEA does not consider such a structure, the interactions between workers is much less structured or controlled. Second, in an IMEA each island represents an individual evolutionary process, sharing some of the same dynamics as standard EAs. In a PEA, however, only a single centralized population exists, samples of which are distributed across workers, but ultimately combined once again in the centralized pool. Therefore, some of the well-known insights derived from IMEA research (regarding, for example, migration policies) are not necessarily relevant in the PEA framework.

Therefore, the recent approach called Randomized Parameter Setting Strategy (RPSS) **??** 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 **??**, that incorporates additional degrees of freedom that must be tuned before performing a run. Such a tuning task can become overwhelming, particularly if the number of islands is excessively large (hundreds). Therefore, the proposal in **?** 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 the EvoSpace PEA, which lacks the fixed population structure of an IMEA.

To gauge the effectiveness of RPSS on a PEA, it is compared with three different parametrization strategies, similar to what is done in **??**. 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, based on the ranges established in Table **??**. 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. However, here we use the Best-Homogeneous approach for direct comparison with **??**. Finally, the random-heterogeneous-parametrization is considered, where the parameters of each worker 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 generator, as in previous sections.

Experiments are carried out using a different number of EvoWorkers N on each problem. The first group of runs are done with $N = 16$ EvoWorkers, and the second with $N = 120$. Based on **??**, 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. while simplifying the tuning process Results are summarized by tracking how the best solution found so far varies with respect to the total number of samples taken from the EvoSpace pool of individuals. These results are presented in Figure 9, where the average performance for each of the three methods evaluated here.

First, for the P-Peaks problem with 16 EvoWorkers. In this case, we can see a clear trend, the random Heterogeneous configuration is very similar with the best homogeneous configuration, depicted in Figure 9(a). This is a promising initial observation, since the heterogeneous configuration did not require any parameter tuning, while the best homogeneous configuration is chosen from a set of 200 runs. Moreover, we see that using an homogeneous configuration with random values achieves noticeably inferior performance. When the number of EvoWorkers is increased, shown in Figure 9(b), a similar trend appears, however the differences among the algorithms is reduced. Nevertheless, it is obvious that using a



(a) 16 EvoWorkers (b) 120 EvoWorkers

Fig. 9 Convergence plots for the P-Peaks with 16 (a) and 120 (b) EvoWorkers .

random heterogeneous parametrization can be used as an off-the shelf approach for EvoSpace parametrization.

4.3 Discussion

5 Concluding Remarks