evospace-js: Asynchronous Pool-Based Execution of Heterogeneous Metaheuristics

Mario García-Valdez
Instituto Tecnolgico de Tijuana
Tijuana BC, Mexico
mario@tectijuana.edu.mx

JJ Merelo
Grupo GeNeura, Depto. ATC + CITIC, Universidad de Granada
Granada, Spain
jmerelo@ugr.es

ABSTRACT

This paper is part of a continuing effort in the field of EC to develop algorithms that follow an opportunistic approach to computing, allowing the exploitation of freely available services over the Internet by using free tiers of cloud services or volunteer computing resources; the EvoSpace model is able to tap from both kind of resources, using asynchronous evolutionary algorithms. We present its design, which follows an an event-driven architecture and asynchronous I/O model, and its implementation, with a server-side tier programmed in Node.js that uses Redis as an in-memory and high performance data store for the population. This population store is exposed to clients running population-based and nature-inspired metaheuristics through a REST API. Additional capabilities where implemented in this version to allow the logging of experiments where heterogeneous algorithms are executed in parallel. These logs can then be transformed to other formats. As a case study an hybrid global optimization algorithm has been implemented mixing two algorithms: a PSO algorithm from the EvoloPy library and a GA using the DEAP framework. The result was transformed to files compatible to the Comparing Continuous Optimizer platform in order to use their post-processing code. Clients in this case have been developed in the Python language, the language used to implement both libraries. The results from this case study suggest, first, that EvoSpace can be used as a paradigm- and language-agnostic platform for population-based optimization algorithms, and also that this software can yield performance improvements and a viable platform to execute and compare asynchronous pool-based metaheuristics.

CCS CONCEPTS

•Computing methodologies → Heuristic function construction;

KEYWORDS

Nature-inspired metaheuristics, Distributed Evolutionary Algorithms

GECCO '17 Companion, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082473

ACM Reference format:
Mario García-Valdez and JJ Merelo. 2017. evospace-js: Asynchronous Pool-Based Execution of Heterogeneous Metaheuristics. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 7 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082473

1 INTRODUCTION

A large body of work exists on the parallelization of EAs, with techniques leveraging multiple CPU cores, many computing nodes, and GPUs [3, 17, 22]. However, asynchronous EAs [1, 2, 20, 23] have started to become common only relatively recently, in an effort to exploit computing resources available through different Internet technologies, including cloud. In this work, we are interested in those asynchronous EAs following an approach that uses a shared *pool* of individuals with a collection of heterogeneous worker processes carry out population search tasks by collaborating through this pool. We will refer to such algorithms as *Pool-based EAs* or PEAs, and highlight the fact that such systems are intrinsically parallel, distributed and asynchronous.

Pool-EAs differ from the Island Model mainly with regards to the responsibilities assigned to the server. When there is a server in the island model, it is responsible for the interaction and synchronization of all the populations, and this server might have a possibly ephemeral *pool* of migrants in the process of being moved from one island to another. In Pool-EAs, on the other hand, the population repository only receives stateless requests from isolated workers or clients. In this way, Pool-EAs are capable of using and leveraging an ad-hoc and ephemeral collaboration of computing resources.

The platform presented in this paper is a new implementation of the EvoSpace model [14] in which workers asynchronously interact with the population pool in the following way. Following their own schedule, EvoWorkers request samples of the population from the pool and perform a local evolutionary search using them as initial population, which after a number of iterations is returned to the pool. The sample size and the number of iterations this cycle lasts depends on the implementation and is actually not so important for the overall performance of the algorithm. Sample size has got more to do with protocol overhead: the bigger the population, the higher proportion of payload-to-total-packet-size is achieved. The number of iterations of every cycle is also mainly an implementation detail: the shorter the cycle, the higher the load on the server. The actual parameters used in this paper are listed in Table 1. This is a particular instance of a Pool-based EA, which, as long as there is a common population pool, leaves every other detail to particular implementations. Other PEAs, for instance,

might return only one individual to the pool, or use it as a read-only resource, not returning any member to it; the *frequency* with which resources are taken or returned to the population are not set by the model either, leaving it as a implementation or model-specific parameter.

The previous version [30] was implemented using CherryPy, a basic HTTP server written in Python. This new version uses Node.js, an event-driven interpreter with a built-in event loop capable of asynchronous I/O [29], that is running on the JavaScript V8 engine. Node.js is used to optimize throughput and scalability of the server.

Additionally to the increased performance this version adds new functionality: In the former version workers could only ask for random samples of a particular size, now clients can retrieve objects from the server ordered by a score. Designers can use this functionality to implement asynchronous versions of the island model or to force the retrieval of different objects in every request resembling a circular queue. Instead of using the JSON-RPC protocol the server functionality is now exposed as a RESTful Web Service. The server now keeps a log of the work performed by workers: The number of evaluations, the best solution in each generation (or iteration), parameters and algorithm used among others. This log can later be used to compare the performance of the algorithm against others, for instance against algorithms using the COCO (COmparing Continuous Optimisers) platform [16]. The aim of the evospace-js software is to provide researchers with a high performance platform in which they can execute pool-based algorithms using heterogeneous workers.

The remainder of the paper proceeds as follows. Section 2 reviews related work. Afterwards, Section 3 describes the proposed EvoSpace implementation, the experimental work is presented in Section 4. Finally, a summary and concluding remarks are given in Section 5.

2 RELATED WORK

There are two important practical issues faced by many EA and other optimization systems, namely the size of the parameter space and the high computational cost when it is compared with mathematical programming or numerical techniques. Concerning the latter, one approach to mitigate this issue is to use parallel or distributed implementations [4, 8]. For instance, Fernández et al. [12] use the well-known Berkeley Open Infrastructure for Network Computing (BOINC) to distribute EA runs across a heterogeneous network of volunteer computers using virtual machines. Another recent example is found in the FlexGP system developed by Sherry et al. [25]. 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. There is a considerable improvement in performance and scalability in this approach, but this scalability has a cost which is proportionally much smaller than installing a permanent infrastructure, but onerous nonetheless.

In general, all the techniques and implementations mentioned above rely on more or less *traditional* parallel or distributed evolutionary algorithms, using *farming* for offloading evaluations to ephemeral resources or using more traditional island-based models in distributed or cloud-based resources. However, there is another approach to distributed EAs: the so called Pool-based architecture [6, 24, 28]. In general, a Pool-based system employs a central repository where the evolving population, or a part of it, is stored. Distributed clients interact with the pool, performing some or all of the basic EA processes (selection, genetic operators, survival), but these clients join the search by just using an API, and quit by simply not doing it any more. Clients are not considered reliable in any way, and the threshold to join the pool and perform an operation is kept as low as possible. A representative work of this approach is that by Merelo et al. [19] implementing a JavaScript based PEA that distributes the evolutionary process over the web, providing the added advantage of not requiring the installation of additional software in each computing node. Other similar cloud-based solutions are based on a global queue of tasks and a Map-Reduce implementation which normally handles failures by the re-execution of tasks [7, 10, 26]. Using the BOINC volunteer platform Smaoui et al. [11] uses work units that consist of a fitness evaluation task and multiple replicas were produced and sent to different clients.

While using a distributed framework can ease the computational cost, it can also exacerbate the first issue mentioned above; i.e., it increases the size of the algorithm parameter space, which makes parameter tuning a more difficult task. The issue of optimal parametrization of EAs is a widely studied subject [5], with many approaches in literature. For instance, one of the most successful approaches is the F-Racing and iterative F-Racing techniques [18]. However, while such algorithms can find high performance parametrization, they require additional computational effort which can be too expensive in some applications (even if they are more efficient than an exhaustive search).

3 EVOSPACE-JS IMPLEMENTATION

The main components of the EvoSpace framework are: the evospace-js population repository, remote clients called EvoWorkers. Each of these components are defined in the following subsections.

3.1 evospace-js Population Repository

The evospace-js server provides a collection of REST methods to operate over a set of objects *ES*, which can be seen as the population. Multiple populations can be created and are distinguished by their identifier ES_id. Objects in each ES_id can be selected, removed or replaced through the following endpoints:

(1) **population_name/initialize** This is a `POST` request used to create a new population.
(2) **population_name/individual** This is a `POST` request used to create and add a new object to a population. The object is defined in a JSON format, and there is no restriction on its structure, only the following properties are required: "id" this is an integer and is generated if not present, "fitness" also defined as a JSON object, the structure been specific to each application, and finally a "chromosome" property again defined as a JavaScript object giving the internal representation of the solution, by default defined as a list of objects. There is also an optional integer property called "score" used when objects are going to be retrieved in a certain order.

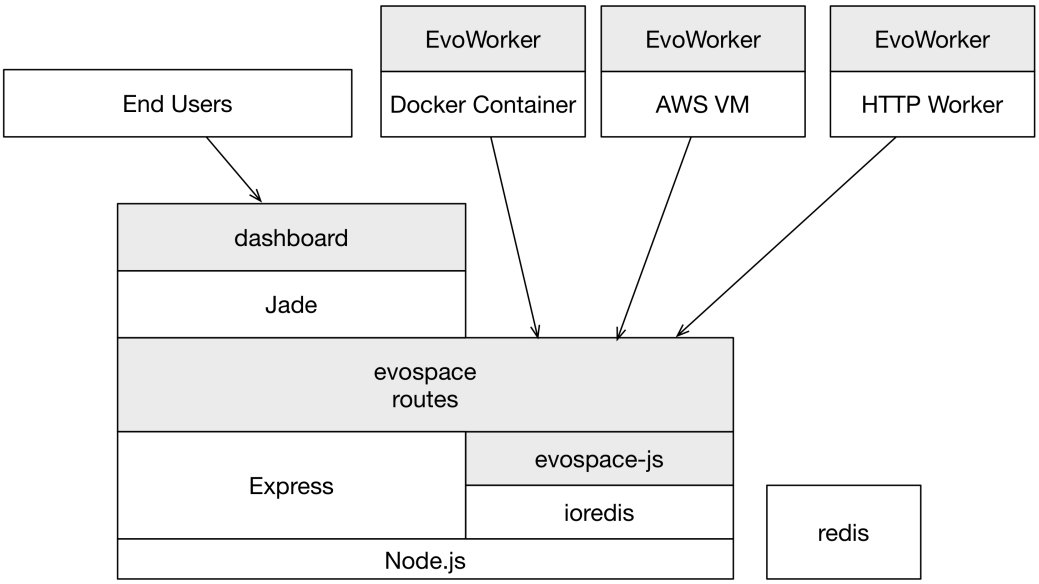

Figure 2: Stack diagram of the evospace-js **framework components.**

(3) **population_name/sample/n** This is a `GET` request used to take from the population a sample of **n** objects. These objects are removed from the population and are no longer available to other requests until and only if they are put back. Objects can be returned to the population either by a `PUT` sample request called from the same client or by a respawn request. The reason for this is to avoid concurrently write conflicts and duplication of work.
(4) **population_name/sample** This is a `POST` request used to put back a sample to the population. The new sample is sent in the request body as a JSON object. If the client created new objects or changed their original state, these objects replace the originals.
(5) **population_name/respawn** This is a `POST` request used to put back **n** samples to their original state. The number of samples is sent in the request body.

There are other secondary REST endpoints used to: select all objects in a population, select objects with scores with in a range, read the top `n` objects according to a score and read the number of objects currently on the population.

The above methods were implemented first as JavaScript library with two classes: `Individual` and `Population` depicted in Figure 1 with calls to the Redis memory store through the **ioredis** asynchronous library.

In order to expose the library as a REST Web service endpoints were implemented using the Express HTTP framework. An optional dashboard type application, can be used to inspect the populations currently available on the server. This dashboard uses the Jade (which has recently been renamed to Pugs) template engine and Express.

When a worker is putting back a sample, it can send an additional property called `benchmark_data` to send supplementary information about the execution of the experiment. This data can later be used to benchmark the performance of the algorithm. This data is again stored in Redis as an ordered list, keeping a log for each experiment. Currently the JSON benchmark data structure contains the following details to later be used by the COCO platform: the algorithm identifier, parameters used, name, dimension, instance and optimal value of the function that is been optimized, worker and experiment identifiers and finally a list of details of each iteration or generation of the local execution. The details include: the best solution and the function value, and the number of function evaluations required. Depending on the application other data could be recorded. The source code for the evospace-js server is in the following Github repository: https://github.com/mariosky/evospace-js.

3.2 EvoWorkers

As we mentioned earlier, EvoWorkers are independent of the population repository, and developers can implement them in any language that supports HTTP requests. To develop an EvoWorker, a programmer could just write the code needed to take a sample of the population and use this sample to replace the initial population of a local algorithm. Then after a certain number of iterations return the current population back to the server.

In this work, EvoWorkers were implemented in Python taking advantage of two open source libraries of nature inspired optimization metaheuristics: DEAP [13], which use the configuration parameters included in Table 2 and EvoloPy [9] using the parameters listed in Table 3. For each algorithm a Python script was responsible for the initialization using the required parameters and setting up the initial population, then after some iterations, the current population and benchmark data is sent back to the server. EvoWorker scripts can run in Docker containers, by receiving the initial parameters as environment variables, and the script ends when it reaches a maximum number of samples. The source code for the Python

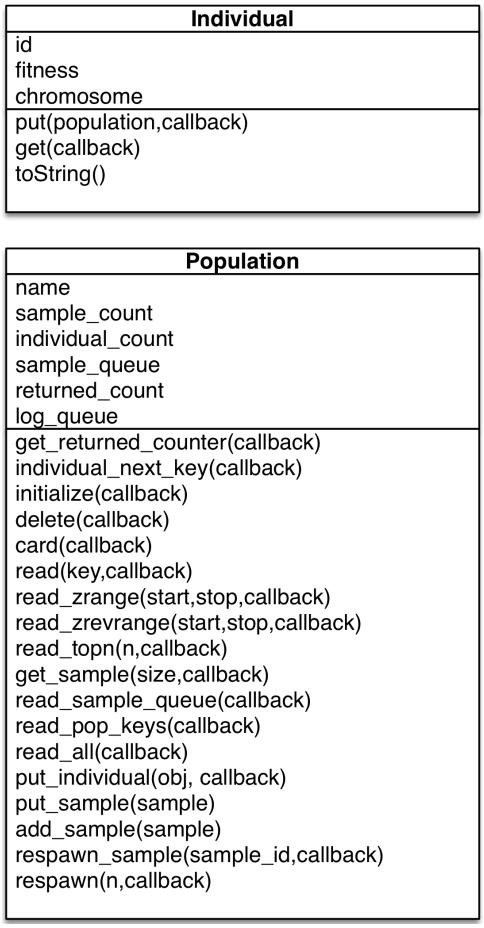

Figure 1: UML Class diagram of the Population and Individual classes.

EvoWorkers proposed in this work are in the following GitHub repository: https://github.com/mariosky/EvoWorker.

4 EXPERIMENTS

As a case study, a simple hybrid algorithm consisting of PSO and GA EvoWorkers was used to run a benchmark that included the first three functions found in the COCO platform: Sphere (F1), Ellipsoid (F2), and Rastrigin (F3). The objective of the algorithm is not to be a competitive solution for the optimization benchmark, as the intention is to test the software functionality. After the execution, a script processed the logs and generated the files needed by the COCO platform post-processing scripts.

A requirement of the COCO platform is that it needs to inspect each function evaluation to keep the log required to analyze the execution. The logging code maintains a sequential record of the number function calls. This exact order is not practical to keep in an asynchronous execution as many workers are calling the function at the same time. For this reason, the granularity of the number of function evaluations and their order is kept at the sample-iteration level. As we mentioned earlier, each worker returns a record with benchmark data, with the number of evaluations performed in each iteration.

As we mentioned earlier, each worker returns the number of evaluations performed in each iteration. The order of function calls was given by the order in which the server received the samples and the order of the iterations in each. On the other hand, the number of function evaluations is incremented in each iteration by the sample size and the best function evaluation is assigned that number, as if in each iteration the best solution was found in the last function evaluation. Instead of increasing the number by one it is incremented by the number of solutions in the sample. Is important to notice that EvoWorkers run the algorithm only for a small number of iterations and with a relatively small sample of the population. For instance, for the COCO benchmark presented in the case study the maximum number of function evaluations in a single iteration was 200.

Table 1: EvoWorker Setup

Dimension	2	3	5	10	20	40
Iterations per Sample	50	50	50	50	50	50
Sample Size	100	100	100	200	200	200
Samples per Worker	20	30	25	25	25	25
PSO Workers	1	1	2	2	4	8

Table 2: DEAP GA EvoWorker Parameters

Search space	$[-4, 4]^D$
Selection	Tournament size=12
Mutation	Gaussian $\mu = 0.0$, $\sigma = 0.5$, indbp=0.05
Mutation Probability	[.1,.6]
Crossover	Two Point
Crossover Probability	[.8,1]

Table 3: EvoloPy PSO EvoWorker Parameters

Search space	$[-4, 4]^D$
V_{max}	6
W_{max}	0.9
W_{min}	0.2
C_1	2
C_2	2

4.1 Set-up

A script was responsible for creating the EvoWorker containers and running the benchmark. The first three functions F1 to F3 were tested with 15 instances for each of the dimensions: 2, 3, 5, 10, and 20. The maximum number of function evaluations was set to $10^5 * dim$. In order to maintain the required number of function evaluations the following EvoWorker setup was set for each dimension. An instance of a function in COCO is a different version of the function, with a different location of the global optimum and a different optimal function value.

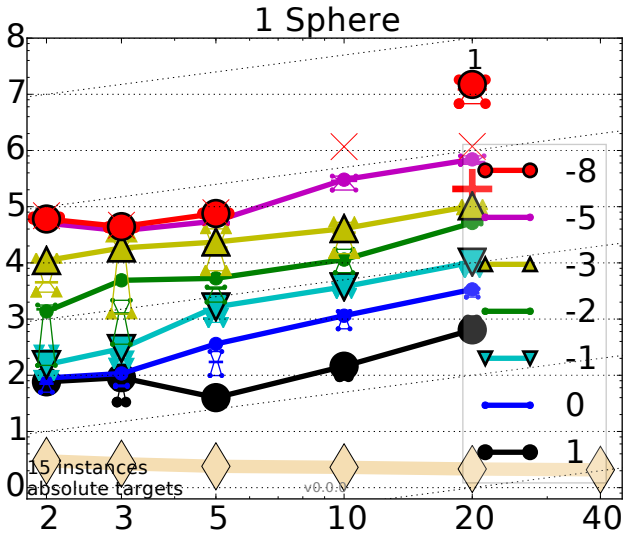

Figure 3: Average numbers of function evaluations to reach the target for every dimension for the Sphere function. This figure and the following ones have been generated with the BBOB report generator. The lines represent the average running times, the crosses the median runtime of successful runs to reach the most difficult target, x-crosses the max number of f-evaluations in any trial. The y scale is logarithmic. The light line on the bottom with diamonds indicates the best algorithm from BBOB 2009. This shows that, for the time being, the runtime is worse than the one obtained in the best BBOB 2009; however, for the time being no attempt to optimize results has been made

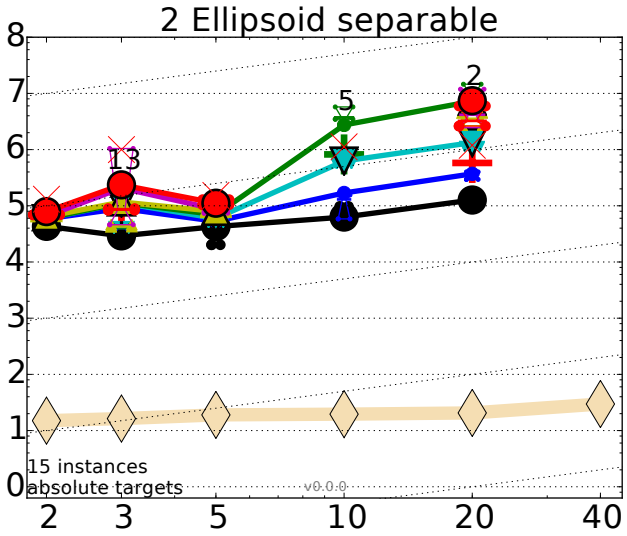

Figure 4: Average numbers of function evaluations to reach target for every dimension for the Ellipsoid function. Interpretation of results as in Figure 3.

4.2 Results

After the experiment was run a script generated the files and folders needed by the COCO post-processing scripts; this script generated the comparative tables and graphics for checking the performance of the algorithm against those found in the COCO repository. Results from experiments according to [16] on the benchmark functions are presented in Figures 3, 5 and 4. The whole experiment, with all dimensions and instances, took less three hours to execute. The results obtained show that the algorithm performs quite well

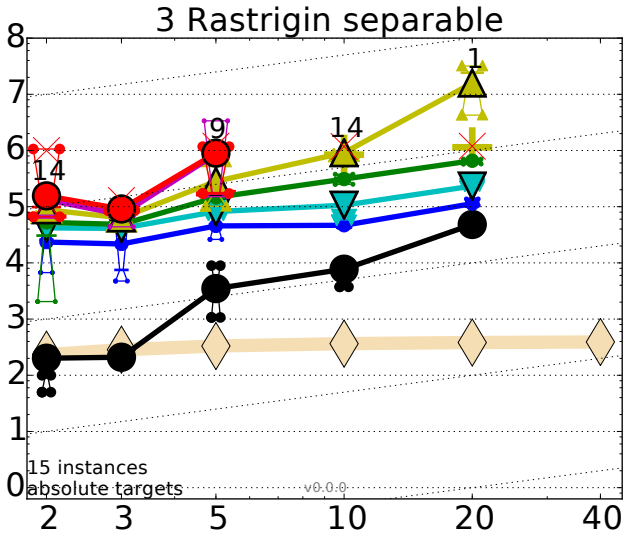

Figure 5: Average numbers of function evaluations to reach target for every dimension for the Rastrigin function. Interpretation of results as in Figure 3.

on separable functions 1-3, both regarding results and scaling when compared to results from other nature inspired algorithms [15]. Results obtained with the Rastrigin function are better than the rest, giving hope that with adequate tuning much better results could be obtained.

In fact, all runs have ended successfully with a completely new asynchronous, heterogeneous, and pool based evolutionary algorithm. Our intention in this paper was to present this framework and present how it can successfully tackle and solve difficult problems using heterogeneous workers that consume the same evospace-js API.

5 CONCLUSIONS AND FURTHER WORK

The evospace-js platform has been applied to implement and test a hybrid nature-inspired algorithm against a testbed of noiseless continuous functions. Design and implementation details have been presented. The results obtained show that an asynchronous execution following a pool-based approach is possible and easy to implement, and that it is able to successfully find solutions to difficult optimization problems. Results, however, are still preliminary and further tuning of the parameters could potentially yield better results, mainly time-wise. There is also the question about the performance of the algorithm on the remaining functions of the benchmark and other real-world problems; however, these are only implementation details.

Our results have been entirely published in GitHub, along with sources. We think it is important to help reproductibility as much as possible by opening our science. Publishing results and software with an open source license will help achieve this reproductibility, and you can find all results, code as well as the source for this paper in https://github.com/mariosky/2017-EvoSpace.

Future lines of work will focus on using other EA or metaheuristic population-based techniques, such as the Grey Wolf Optimizer [21] or Differential Evolution [27] for creating workers that are heterogeneous in more than one sense. RPSS could be used in those cases where each algorithm has a different set of parameters, but also to randomly select the technique employed in each node.

Another interesting line of work is the dynamic adaptation of parameters by measuring the diversity of each worker or returned sample. This adaptation could be especially useful in cases where the random parametrization technique seems to achieve bad results.

ACKNOWLEDGMENTS

This work has been supported in part by Ministerio español de Economía y Competitividad under project TIN2014-56494-C4-3-P (UGR-EPHEMECH).

REFERENCES

[1] Enrique Alba and José M Troya. 2001. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems* 17, 4 (2001), 451–465.

[2] J. Atienza, P. A. Castillo, M. García, J. González, and J.J. Merelo. 2000. Jenetic: a distributed, fine-grained, asynchronous evolutionary algorithm using Jini. In *Proc. JCIS 2000 (Joint Conference on Information Sciences)*, P. P. Wang (Ed.), Vol. I. 1087–1089. ISBN: 0-9643456-9-2.

[3] Erick Cantu-Paz. 2000. *Efficient and accurate parallel genetic algorithms*. Vol. 1. Springer Science & Business Media.

[4] Erick Cantú-Paz. 2001. Migration Policies, Selection Pressure, and Parallel Evolutionary Algorithms. *Journal of Heuristics* 7, 4 (2001), 311–334. DOI: http://dx.doi.org/10.1023/A:1011375326814

[5] Kenneth De Jong. 2007. Parameter setting in EAs: a 30 year perspective. In *Parameter setting in evolutionary algorithms*. Springer, 1–18.

[6] P.S. de Souza and S.N. Talukdar. 1991. Genetic algorithms in asynchronous teams. In *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 392–399.

[7] Sergio Di Martino, Filomena Ferrucci, Valerio Maggio, and Federica Sarro. 2013. Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud. In *Software Testing in the Cloud: Perspectives on an Emerging Discipline*. IGI Global, IGI Global, 113–135. DOI: http://dx.doi.org/10.4018/978-1-4666-2536-5.ch006

[8] Jerzy Duda and Wojciech Dlubacz. 2013. GPU acceleration for the web browser based evolutionary computing system. In *System Theory, Control and Computing (ICSTCC), 2013 17th International Conference*. IEEE, 751–756.

[9] Hossam Faris, Ibrahim Aljarah, Seyedali Mirjalili, Pedro A. Castillo, and Juan J. Merelo. 2016. EvoloPy: An Open-source Nature-inspired Optimization Framework in Python. In *Proceedings of the 8th International Joint Conference on Computational Intelligence, IJCCI 2016, Volume 1: ECTA, Porto, Portugal, November 9-11, 2016.*, Juan Julián Merelo Guervós, Fernando Melício, José Manuel Cadenas, António Dourado, Kurosh Madani, António E. Ruano, and Joaquim Filipe (Eds.). SciTePress, 171–177. DOI: http://dx.doi.org/10.5220/0006048201710177

[10] Pedro Fazenda, James McDermott, and Una-May O'Reilly. 2012. A Library to Run Evolutionary Algorithms in the Cloud Using Mapreduce. In *Proceedings of the 2012T European Conference on Applications of Evolutionary Computation (EvoApplications'12)*. Springer-Verlag, Berlin, Heidelberg, 416–425. DOI: http://dx.doi.org/10.1007/978-3-642-29178-4_42

[11] Malek Smaoui Feki, Viet Huy Nguyen, and Marc Garbey. 2009. Parallel Genetic Algorithm Implementation for BOINC. In *PARCO (Advances in Parallel Computing)*, Barbara M. Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, Frans J. Peters, and Thierry Priol (Eds.), Vol. 19. IOS Press, 212–219.

[12] Francisco Fernández De Vega, Gustavo Olague, Leonardo Trujillo, and Daniel Lombraña González. 2013. Customizable Execution Environments for Evolutionary Computation Using BOINC + Virtualization. *Natural Computing* 12, 2 (2013), 163–177.

[13] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research* 13, Jul (2012), 2171–2175.

[14] Mario García-Valdez, Leonardo Trujillo, Juan-J Merelo, Francisco Fernández de Vega, and Gustavo Olague. 2015. The EvoSpace Model for Pool-Based Evolutionary Algorithms. *Journal of Grid Computing* 13, 3 (2015), 329–349. DOI: http://dx.doi.org/10.1007/s10723-014-9319-2

[15] Anne Augerfi!?Steffen Finckfi!?Nikolaus Hansen and Raymond Ros. 2010. BBOB 2009: Comparison Tables of All Algorithms on All Noiseless Functions. (2010).

[16] Nikolaus Hansen, Anne Auger, Olaf Mersmann, Tea Tusar, and Dimo Brockhoff. 2016. COCO: a platform for comparing continuous optimizers in a black-box setting. (2016).

[17] Johannes Hofmann, Steffen Limmer, and Dietmar Fey. 2013. Performance investigations of genetic algorithms on graphics cards. *Swarm and Evolutionary Computation* 12 (2013), 33–47.

[18] Manuel López-Ibánez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. 2011. *The irace package, iterated race for automatic algorithm configuration*. Technical Report. Citeseer.

[19] J. J. Merelo, P.A. Castillo, J.L.J. Laredo, A. Mora, and A. Prieto. 2008. Asynchronous Distributed Genetic Algorithms with JavaScript and JSON. In *WCCI 2008 Proceedings*. IEEE Press, 1372–1379. http://atc.ugr.es/I+D+i/congresos/2008/CEC_2008_1372.pdf

[20] Juan J. Merelo, Antonio M. Mora, Pedro A. Castillo, Juan L. J. Laredo, Lourdes Araujo, Ken C. Sharman, Anna I. Esparcia-Alczar, Eva Alfaro-Cid, and Carlos Cotta. 2008. Testing the Intermediate Disturbance Hypothesis: Effect of Asynchronous Population Incorporation on Multi-Deme Evolutionary Algorithms. In *Parallel Problem Solving from Nature - PPSN X (LNCS)*, Gunter Rudolph, Thomas Jansen, Simon Lucas, Carlo Poloni, and Nicola Beume (Eds.), Vol. 5199. Springer, Dortmund, 266–275. DOI: http://dx.doi.org/10.1007/978-3-540-87700-4_27

[21] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. 2014. Grey wolf optimizer. *Advances in Engineering Software* 69 (2014), 46–61.

[22] Heinz Mühlenbein. 1989. Parallel genetic algorithms, population genetics and combinatorial optimization. In *Workshop on Parallel Processing: Logic, Organization, and Technology*. Springer, 398–406.

[23] Victor M. Rivas, Juan Julián Merelo-Guervós, Gustavo Romero-López, Maribel Arenas-García, and Antonio M. Mora. 2014. An Object-Oriented Library in JavaScript to Build Modular and Flexible Cross-Platform Evolutionary Algorithms. In *Applications of Evolutionary Computation*, Anna I. Esparcia-Alcázar and Antonio M. Mora (Eds.). Springer Berlin Heidelberg, 853–862. DOI: http://dx.doi.org/10.1007/978-3-662-45523-4_69

[24] G. Roy, Hyunyoung Lee, J.L. Welch, Yuan Zhao, V. Pandey, and D. Thurston. 2009. A distributed pool architecture for genetic algorithms. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*. 1177–1184. DOI: http://dx.doi.org/10.1109/CEC.2009.4983079

[25] Dylan Sherry, Kalyan Veeramachaneni, James McDermott, and Una-May O'Reilly. 2012. Flex-GP: genetic programming on the cloud. In *Applications of Evolutionary Computation*. Springer, 477–486.

[26] Dylan Sherry, Kalyan Veeramachaneni, James McDermott, and Una-May OfiReilly. 2012. Flex-GP: Genetic Programming on the Cloud. In *Applications of Evolutionary Computation*, Cecilia Chio, Alexandros Agapitos, Stefano Cagnoni, Carlos Cotta, FranciscoFernndez Vega, GianniA. Caro, Rolf Drechsler, Anik Ekrt, AnnaI. Esparcia-Alczar, Muddassar Farooq, WilliamB. Langdon, JuanJ. Merelo-Guervs, Mike Preuss, Hendrik Richter, Sara Silva, Anabela Simes, Giovanni Squillero, Ernesto Tarantino, AndreaG.B. Tettamanzi, Julian Togelius, Neil Urquhart, A.ima Uyar, and GeorgiosN. Yannakakis (Eds.). Lecture Notes in Computer Science, Vol. 7248. Springer Berlin Heidelberg, 477–486. DOI: http://dx.doi.org/10.1007/978-3-642-29178-4_48

[27] Rainer Storn and Kenneth Price. 1997. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization* 11, 4 (1997), 341–359.

[28] S. Talukdar, L. Baerentzen, A. Gove, and P. De Souza. 1998. Asynchronous teams: Cooperation schemes for autonomous agents. *Journal of Heuristics* 4, 4 (1998), 295–321.

[29] Stefan Tilkov and Steve Vinoski. 2010. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 14, 6 (2010), 80–83.

[30] Mario García Valdez, Leonardo Trujillo, Juan Julián Merelo-Guervós, Francisco Fernández de Vega, and Gustavo Olague. 2015. The EvoSpace Model for Pool-Based Evolutionary Algorithms. *J. Grid Comput.* 13, 3 (2015), 329–349. DOI: http://dx.doi.org/10.1007/s10723-014-9319-2