2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing

Pool vs. island based evolutionary algorithms: an initial exploration

J.J. Merelo, A.M. Mora, C. M. Fernandes

University of Granada
Department of Computer Architecture and Technology
ETSIIT- 18071 - Granada (Spain)
Email: {jmerelo,amorag,cfernandes}@geneura.ugr.es

Anna I. Esparcia-Alcázar
S2 Grupo
Valencia, Spain
Email: aesparcia@s2grupo.es

Juan L. J. Laredo
University of Luxembourg
Email: juan.jimenez@uni.lu

***Abstract*—This paper explores the scalability and performance of pool and island based evolutionary algorithms, both of them using as a mean of interaction an object store; we call this family of algorithms SofEA. This object store allows the different clients to interact asynchronously; the point of the creation of this framework is to build a system for spontaneous and voluntary distributed evolutionary computation. The fact that each client is autonomous leads to a complex behavior that will be examined in the work, so that the design can be validated, rules of thumb can be extracted, and the limits of scalability can be found. In this paper we advance the design of an asynchronous, fault-tolerant and scalable distributed evolutionary algorithm based on the object store CouchDB. We test experimentally the different options and show the trade-offs that pool and island-based solutions offer.**

***Keywords*-Cloud services, distributed evolutionary computation, evolutionary algorithms, pool-based computing.**

I. Introduction

Most studies on evolutionary algorithms (EAs) rely on traditional execution environments with single memory and CPU. These environments can be studied traditionally and extended to parallel and even distributed environments, provided that there are certain conditions, such as synchrony, homogeneity and centralized operations, for instance. However, in the last few years the range of possible computational environments has been extended greatly, to the point that it is possible to achieve a bigger computational raw power [1] by creating ad hoc, loosely linked, and heterogeneous frameworks where EAs can be run. One of such targets are the so-called *volunteer computing* or *desktop grid* environments, [2], [3], which have been used extensively so far in evolutionary algorithms, for instance in [4]–[6].

In this paper we present a system whose main objective is to adapt an evolutionary algorithm to a volunteer computing environment; an evolutionary algorithm is a population-based algorithm which evolves sets of solutions inspired by the principles of biological evolution [7]. The basic evolutionary algorithm loop consists in evaluating a set of solutions, assigning a *fitness* to each one, and them choosing them for reproduction based on that fitness, changing them (performing *mutation*) or combining them (doing *crossover*) to create a new generation; this is done until the solution is reached or during a predetermined amount of steps (which are called *generations*).

Environments for *volunteer* evolutionary computation have been implemented in several ways, from a farming approach (farming out evaluations to clients) to an island-like approach (which was the one used in [8], [9]). Both present problems from the *ad-hoc* framework point of view. The first one has to run the EA on the sever, which is not too pliable to a client-server architecture and makes the server a bottleneck; the second one is better, performance-wise, but the server does not keep track of a good part of the population (just the ones that are being migrated among islands). However, it is an interesting architecture that will be one of the possible options we will test in this paper.

The other architectures we will test here are pool-based. The basic idea is that the bulk (or possibly all) the population, is kept in a server with some structure. The clients, which can join and leave at any time, pull a set of tasks from the server, perform them, and return the result. If we identify a *task* with a single individual, this is as spontaneous as it gets: a client can perform a single step and leave the experiment without any further consequences, since the state is kept on the server.

An island-based system, on the other hand, is less amenable to spontaneous collaboration since the server usually keeps only track of migrant individuals. Even in this kind of systems there are several possible ways of configuring them, that is why in this paper we will test several configurations, mainly measuring their scaling capability with respect to the number of nodes and performance, since *spontaneity* is mainly a requisite which pool-based systems have in a high degree and island-based in a low degree.

After presenting in other papers [10], [11] several versions of SofEA, a pool-based evolutionary algorithm, in this paper we introduce new versions and compare its scalability and performance with an island-based evolutionary algorithm that uses the same infrastructure. This version is more suitable for using it in real-life applications. SofEA could run in browsers using the embedded JavaScript interpreter, and, using this, massive experiments via volunteers using it by simply visiting an URL.

The rest of the paper is structured as follows. In the next section the state of the art in these topics is presented; after it, section III introduces CouchDB, the document store we are going to consider for mapping the EAs. The

978-0-7695-4841-8/12 $26.00 © 2012 IEEE
DOI 10.1109/3PGCIC.2012.56

CouchDB-based evolutionary algorithms are described in section IV, and the experiments performed for testing them are included in section V. Finally, the conclusions and future lines of work are presented in section VI.

II. State of the Art

In this section we will examine pool-based distributed computing systems, mainly those that have been applied to evolutionary algorithms. The most popular model for asynchronous distributed algorithms is called *A-teams*, where A stands for *asynchronous* [12]. A-Teams combine different algorithms that share a memory in closed loops and are a way of specifying data flow among different methods to solve a problem. A-Teams are not intrinsically evolutionary methods but have been successfully applied in the last decades to a wide variety of problems [13]; their authors have released a toolkit that can be used to implement solutions to different problems. A-Teams can be implemented in many different ways, but they often refer to a *pool* or shared memory from which solutions (or sets of them) can be drawn, improved and put back, or to where newly constructed solutions can be shared among all the agents participating in the experiment.

Taking then one step down and entering the realm of the implementations (away from the models exposed above), several authors have directly implemented evolutionary algorithms in a pool based architecture, where the basic idea is to use a (more or less persistent) store of solutions from which the evolutionary algorithm draws its individuals, instead of having the population as a data structure that is taken from one method to the next. The first papers in the 90s used shared memory systems such as Linda [14]. Lately, multi-threaded systems with a shared memory [15] have been proposed; this memory can be read from all threads, but is divided in chunks writable by only one of the threads. Relational database systems [16] have also been used, proving their capability for avoiding algorithms with explicit synchronization and their fault-tolerance, at least to client failure, providing a persistent storage for population from which solutions can be, later on, retrieved. A database is, for instance, used in Distributed BEAGLE [17], which separates evolution and evaluation with a single *evolver* client independent from the *evaluator* clients, both working with a central database.

Even if the database is a single point of failure, this can be avoided by replication; besides, the state of evolution is partially held by anyone of the clients at a particular moment, so even in the event of a database failure all the information is not lost.

III. Brief introduction to CouchDB

CouchDB is a key-value store [18] that uses JSON (JavaScript Object Notation, a text serialization of arbitrary data structures [19]) for expressing them, being able to store any kind of data structure, called *documents* in this context. Objects can be retrieved by key or range of keys directly, but complex queries using map/reduce [20] operations, written by default in JavaScript and called *views*, can be applied to them. *Map* operations apply individually to each element in the database, while *reduce* ones are applied to lists of keys and values resulting from the map operation. For instance, if we want to count how many documents have a particular attribute, a *map* operation would *emit* that attribute as key, with the document itself (or any other attribute) as value. The *Reduce* operation would count the number of keys with the same value.

CouchDB uses a simple REST (Representational State Transfer) application programming interface (API) that can be accessed either from the command line or from multitude of client libraries; this API can be used either to access objects directly or to apply operations to them. Every document in the database is provided with several additional attributes, the most important of which will be for us the *revision*, a versioning attribute that changes every time an object is modified; revisions take the form `1-91285b0279dc582d8e1549c84c9c1406`) and its main part increases every update.

The easiest and fastest, not to mention highly concurrent, operations in CouchDB are those that involve querying using keys. Inserting or updating a set of elements in bulk is very fast too. More complex queries involving document content, that is, *views*, that include map/reduce operations, are slower and cannot be done concurrently to such a degree; that is why it is better to design high-performance applications around the use of keys and reduce use of views as much as possible. However, there is no other way of accessing the *content* of the documents stored in the database, that is why they are in many cases unavoidable.

Apart from technical reasons, one of the advantages of CouchDB is its wide availability for all common operating system, this means that one can develop for CouchDB anywhere. Moreover, since its API is based in the easy-to-build REST convention, clients can also be written in most common computer languages; it can even be used from the command line composing URLs by hand. Ultimately, building a CouchDB client or even a client-server application is straightforward and, in order to use it in a volunteer computing environment, enables support of fitness functions and algorithms written in virtually any language, even an mix of several. Quite importantly, CouchDB is an one-stop framework for developing web applications, that is, clients can be completely embedded in the browser using JQuery and JavaScript.

CouchDB is able to support thousands of concurrent users; the maximum reported quantity of concurrent users is 2300[1]. This is more than enough for supporting a long range of single-server scalability. If a higher number of concurrent users is previewed, two-way replication can be easily set up with CouchDB. On the other hand, for documents of the size we are handling (several Ks) CouchDB can serve several requests per second, with updates of dozens of documents in a single request. A

[1] http://nosql.mypopescu.com/post/9891985838/help-couchdb-break-the-c10k-barrier

request for a single document (done in bulk) would be in the ballpark of a millisecond.

Using CouchDB is also easy; each database has an entry URL of the form `http://host.com:5984/database`; database name and host or this URL are the only parameters needed to access the database. Since they are configured by default to be accessed only from the local host, it is not usual to include authentication methods, although these can be added if needed (not in the application presented here).

Eventually, the models shown in this paper might be adapted to other data stores such as MongoDB, Cassandra or Riak [21], specially this last one, whose features are very similar to those of CouchDB. However, in this paper we will focus in the development of evolutionary algorithms that fit CouchDB architecture. This will be shown next.

IV. Pool and island evolutionary algorithms based on CouchDB

Several versions of SofEA have already been presented in [10], [11] and evolved in [22]. The models examined in this paper are different from previously published ones [10], [11], achieving greater speeds and reducing the design space while maintaining the fault-tolerance and asynchrony of the pool-based architecture.

There is a single thing all SofEA algorithms have in common: a population kept in a pool from where it is drawn by the clients, operated on and put back in it. Latest version, introduced in [22], worked as follows: after an initial set of evaluated chromosomes were created, clients took a block of individuals and applied a single evolutionary algorithm generation on them, putting results back in the population as many as were deleted from it. In order to keep the population constant, the program kept track of the conflicts (individuals already present in the population) and eliminated as many from the pool in the next iteration. That avoided population explosions, and had a benefit on the population. Individuals were stored in a document (items in the CouchDB database are called *documents*, see the previous section) that included the chromosome, a random constant and its fitness, and using its binary string as key. We will call this version BaseSofEA.

However, even if that version solved several problems, it still presented an obstacle to scalability: the need to keep the population constant, which was not adaptable to an increasing number of clients and, even its performance was much better with this version than with the previous ones for a single client. New clients did not add speed, but reliability, that is, likelihood of reaching the solution in a fixed time.

That is why we introduced a new version of SofEA which completely eliminated the concept of *living population*. The concept is quite simple: instead of working with a *live* and *dead* population, the former to be kept constant to avoid decreasing the selective pressure, the *live* population (the one the genetic operators will act on) is simply a set of the best chromosomes ranked by fitness. This algorithm, which we will call *EliteSofEA*, works in the following way:

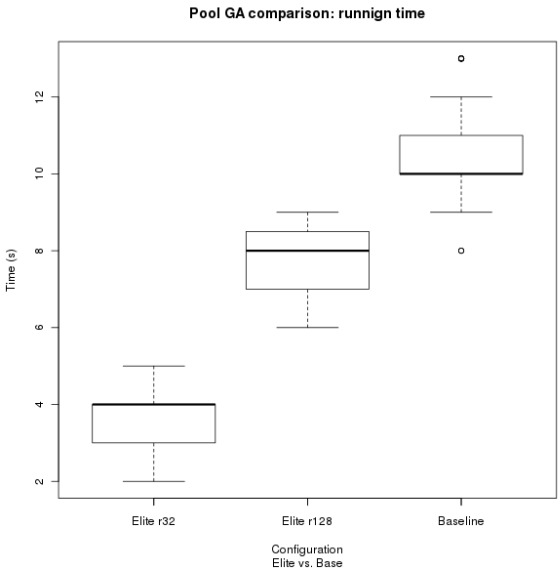
1) Generate an initial population of size p
2) While the solution has not been reached,
    a) Obtain p individuals from the population, record the worst fitness of the set to use it later on as a cut-off for sending new individuals to the pool.
    b) Apply a single generation to this population.
    c) Put back in the pool the individuals that are better than the worst incoming individual.

Other small improvements were also added to this version, tapping from our experience using the system: the document that stored the chromosome just included the fitness, eliminating the need for a random constant and included the chromosome only as key; this resulted in a more efficient storage but also in faster operations when retrieving or updating the database. This made this version the fastest of the pool-based ones, as shown in figure 1. EliteSofEA works always on the individuals with the best fitness, having thus the higher selective pressure of the group of algorithms designed so far. This probably explains the results shown in the figure, that indicate that the speed is mainly due to the reduction in the number of evaluations to solution, implying an algorithmic advantage over the old versions. This is the main reason for including this new approach for comparison with island EAs, instead of the ones introduced in [22].

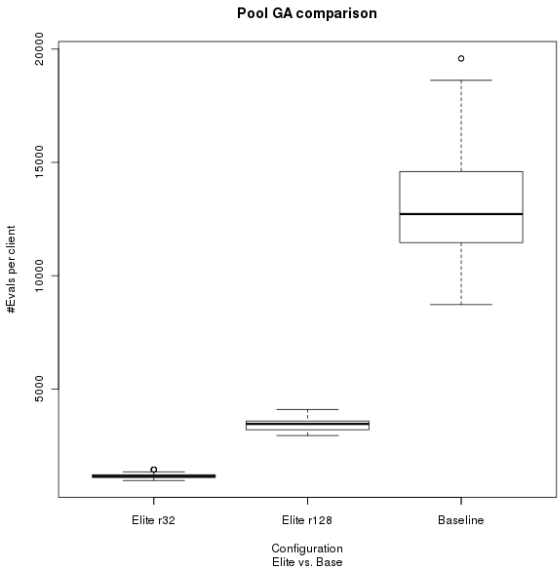
The last algorithm to be tested in this paper is essentially an island GA [23], which is why we will call it IslandSofEA. Every island runs an evolutionary algorithm independently, but, after a certain number of generations, it obtains from the pool the best n individuals, putting back in it the best n individuals in the latest generations. The evolutionary algorithm in every island also uses a rank-based policy for insertion of new individuals: every generation, only the best distinct p individuals are kept. This guarantees that the population holds always different individuals, maintaining diversity. Every generation, the client checks whether the solution has been found, stopping if any of the clients finds it. The main difference between this an other island-based GAs is that the pool acts as an interconnection grid, connecting all islands to every other, but without needing explicit connections or synchronization.

V. Experiments and results

In order to compare the different algorithms, we have used MaxOnes, that is, maximizing the number of ones in a binary string, with several lengths, up to 256 bits; however, results with lengths smaller than 256 did not offer much differentiation among the algorithms, since they were solved too quickly by all of them, so we have mainly used this bigger length to reach conclusions in this paper. Since the main factor influencing results is the time it takes to evaluate fitness and the population



(a) Running time (in seconds)



(b) Number of evaluations

Figure 1. Boxplots comparing BaseSofEA and EliteSofEA, running time (left) and number of evaluations (right). Two different versions of this one have been tested, with client block size equal to 32 (left) and 128 (middle).

size needed to solve it, we wanted to concentrate on one in which this speed was very small with respect to application latency, so that the pure algorithmic features of the proposed system are emphasized. Problems that need bigger population sizes will increase the range in which they scale, but in principle it needs not affect differentially the two types of algorithms shown here; that is why we consider the set of experiments offered here enough to reach meaningful conclusions.

We tested all the algorithms with an initial population that was divided among the clients when its number was increased, that is, experiments were made with *constant* population.

IslandSofEA performed migration after 25 generations. In fact, asynchronous algorithms combined with pool-based ones do not have a clear sense of *population*, which is a global concept, but we did this in order to make conditions for all algorithms as close as possible. Experiments were repeated 30 times on a Ubuntu 11.04 computer and CouchDB 1.0.1. All programs, parameters and results are available under a GPL licence from http://goo.gl/nhon7. Clients were written in Perl and used the `Algorithm::Evolutionary::Simple` module, which is available from CPAN; clients were running on the same computer, which actually did not result in an excessive workload.

One of our main objectives with these experiments was to measure scaling, and the influence of the configuration in the behavior of the algorithm. This is plotted in figure 2. This figure shows that the number of evaluations *per client* is different for both algorithms. It scales approximately in the same way, but the number of evaluations for EliteSofEA is smaller except in the single-client situation (left-most box). This should be expected since EliteSofEA always uses the best set of individuals in the pool to apply a single generation; IslandSofEA obtains the latest global best only in the generation after migration is performed. In principle, this would imply EliteSofEA to be faster; however, since it does a bigger amount of request to CouchDB, and one per generation, it is actually much slower, with an average of 1.173 seconds for IslandSofEA vs. 12.19 for EliteSofEA for 8 clients and population size 32, 2.067 and 30.43 for a single client and population = 256. This also shows that scaling is better for EliteSofEA: 2.5 vs 1.7 when the number of clients increases from 1 to 8. Of course, IslandSofEA is an order of magnitude faster than EliteSofEA, so this scaling is eclipsed by the raw speed of the former.

Results for problems with a smaller size, like MaxOnes with 200 bits, are similar, we show in figure 3 the averages for the IslandSofEA, comparing it with the results shown above. Scaling is similar in both cases, with evaluations for 8 clients (and population divided by 8) around 1/3 of those needed for a single client. These numbers of evaluations translate more or less linearly to the time to solution, hinting at a scaling that, while being good, is not even lineal. However, that was not an objective of this algorithm; in a volunteer computation context, we only seek an increase in speed, even small, when new clients are added.

VI. Conclusions, Discussion and Future Work

In this work we introduce a pool-based version of the SofEA pool-based evolutionary algorithm, together with an island-based EA that uses the same framework. The first conclusion is that the pool-based algorithm presented in this paper, EliteSofEA, shows the best number of evaluations, speed and scalability from the set of SofEA

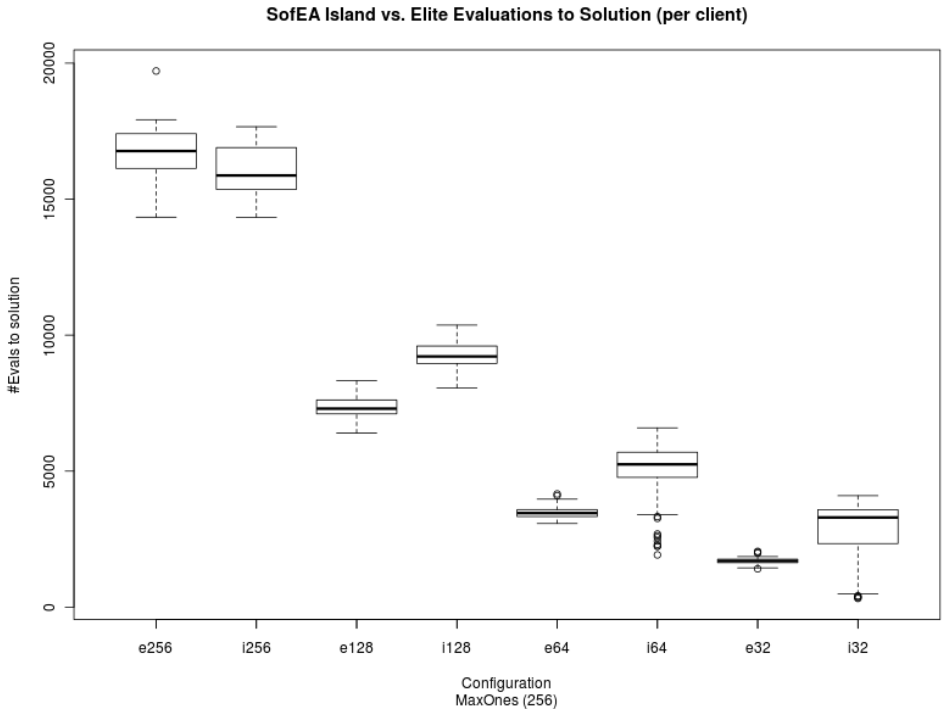


Figure 2. Boxplots comparing the number of evaluations for IslandSofEA and EliteSofEA. The x axis shows the client population size p; the number of clients is 256/p. The initial indicates the type of algorithm, e for Elite and i for Island.

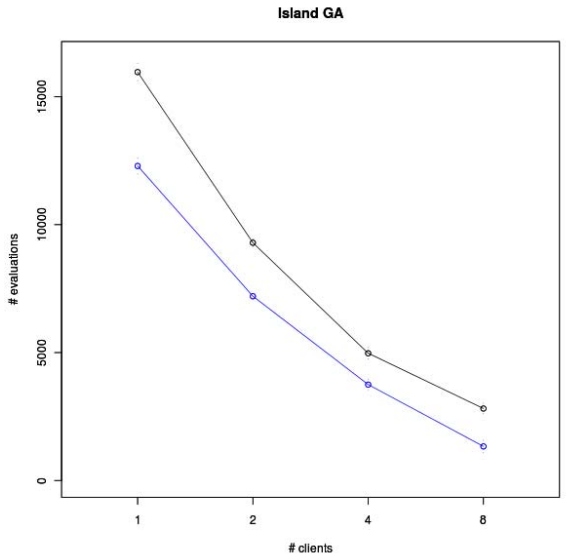


Figure 3. Average number of evaluations for IslandSofEA solving MaxOnes with 256 (black or dark) and 200 (blue or light).

algorithms examined so far [10], [11], [22]. This is an interesting result by itself, and has been proved by a whole set of experiments and problem sizes (not shown here). For problems which are not deceptive and which need a pool-based algorithm, EliteSofEA is the best option. In this paper we have tested two algorithms that use CouchDB as a data store, one that uses this store as a pool (EliteSofEA) and another that uses it as a simple store to interchange individuals between individuals (IslandSofEA). Experiments show that EliteSofEA needs fewer evaluations to reach the solution and its scalability is better when adding new clients; however, IslandSofEA is much faster since its use of high-latency database requests is lower.

This is largely the result of experiments with a single parametrization; however, EliteSofEA is largely parameter-less, and IslandSofEA could be parametrized for the kind of algorithms in every island and the number of generations to migration. However, even if this quantity is tuned, we do not foresee a big influence in the above said conclusions. In static experiments where we know in advance the number of clients and how long they will be staying, IslandSofEA and, in general, island-based evolutionary algorithms are the best option. However, in environments with spontaneous addition/vanishing of clients where these are expected to contribute a single transaction, EliteSofEA or a pool-based EA is the best option. Taking into account that a pool-based architecture can support both types of algorithms (even at the same time) and offers advantages such as persistence and asynchronous operation, we conclude that pool-based architectures represent a very good option that should be explored further.

In the future, it would be interesting to test the system with more heavy-duty problems, such as MMDP or P-

Peaks, whose solution requires a bigger populations for chromosomes with the same size, and thus a higher number of evaluations to reach the solution, but also a fitness function that takes longer to evaluate. This will allow us to evaluate a higher range of number of nodes and check when the physical limits of number of clients is reached.

ACKNOWLEDGMENTS

This work is supported by projects TIN2011-28627-C04-02 awarded by the Spanish Ministry of Science and Innovation, P08-TIC-03903 awarded by the Andalusian Regional Government and Luxembourg FNR GreenIT Project (C09/IS/05). We are also grateful to the reviewers, for helpful comments.

REFERENCES

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: an experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[2] F. Soares, L. Silva, and J. Silva, "How to get volunteers for web-based metacomputing," in *In Proc. of the Distributed Computing on the Web (DCW98), Germany.* Citeseer, June 1998, pp. 264–276.

[3] A. Holohan and A. Garg, "Collaboration online: The example of distributed computing," *Journal of Computer-Mediated Communication*, vol. 10, no. 4, p. 23, 2005.

[4] D. L. Gonzalez, F. F. de Vega, L. Trujillo, G. Olague, F. C. de la O, M. Cardenas, L. Araujo, P. A. Castillo, and K. Sharman, "Increasing GP computing power via volunteer computing," *CoRR*, vol. abs/0801.1210, 2008.

[5] D. Gonzalez, F. de Vega, L. Trujillo, G. Olague, L. Araujo, P. Castillo, J. Merelo, and K. Sharman, "Increasing GP computing power for free via desktop GRID computing and virtualization," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, Feb. 2009, pp. 419–423.

[6] T. Desell, B. Szymanski, and C. Varela, "An asynchronous hybrid genetic-simplex search for modeling the Milky Way galaxy using volunteer computing," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, ser. GECCO '08. New York, NY, USA: ACM, 2008, pp. 921–928. [Online]. Available: http://doi.acm.org/10.1145/1389095.1389273

[7] D. Fogel, "What is evolutionary computation?" *Spectrum, IEEE*, vol. 37, no. 2, pp. 26–28, 2000.

[8] J. J. Merelo, P. Castillo, J. Laredo, A. Mora, and A. Prieto, "Asynchronous distributed genetic algorithms with Javascript and JSON," in *WCCI 2008 Proceedings*. IEEE Press, 2008, pp. 1372–1379. [Online]. Available: http://atc.ugr.es/I+D+i/congresos/2008/CEC_2008_1372.pdf

[9] J.-J. Merelo-Guervós, M. G. Arenas, A. M. Mora, P. A. Castillo, G. Romero, and J. L. J. Laredo, "Cloud-based evolutionary algorithms: An algorithmic study," *CoRR*, vol. abs/1105.6205, 2011.

[10] J.-J. Merelo-Guervós, A. Mora, J. A. Cruz, and A. I. Esparcia, "Pool-based distributed evolutionary algorithms using an object database," in *EvoApplications 2012 Proceedings*, C. di Chio et al., Ed., 2012, pp. 441–450.

[11] J.-J. Merelo Guervos, A. M. Mora, J. A. Cruz Almaguer, A. I. Esparcia-Alcazar, and C. Cotta, "Scaling in distributed evolutionary algorithms with persistent population," in *CEC 2012 Proceedings*, 2012, accepted, to be published.

[12] S. Talukdar, S. Murthy, and R. Akkiraju, "Asynchronous teams," *International Series in Operations Research and Management Science*, pp. 537–556, 2003.

[13] P. Jedrzejowicz, "A-teams and their applications," in *Computational Collective Intelligence. Semantic Web, Social Networks and Multiagent Systems*, ser. Lecture Notes in Computer Science, N. Nguyen, R. Kowalczyk, and S.-M. Chen, Eds. Springer Berlin / Heidelberg, 2009, vol. 5796, pp. 36–50. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04441-0_3

[14] M. Davis, L. Liu, and J. Elias, "VLSI circuit synthesis using a parallel genetic algorithm," in *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, jun 1994, pp. 104 –109 vol.1.

[15] G. Roy, H. Lee, J. Welch, Y. Zhao, V. Pandey, and D. Thurston, "A distributed pool architecture for genetic algorithms," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, May 2009, pp. 1177–1184.

[16] A. Bollini and M. Piastra, "Distributed and persistent evolutionary algorithms: a design pattern," in *Genetic Programming, Proceedings EuroGP'99*, ser. Lecture notes in computer science, no. 1598. Springer, 1999, pp. 173–183.

[17] C. Gagné, M. Parizeau, and M. Dubreuil, "Distributed BEAGLE: An environment for parallel and distributed evolutionary computations," in *Proc. of the 17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS)*, vol. 2003, 2003, pp. 201–208.

[18] J. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide*. Oreilly & Associates Inc, 2009.

[19] D. Crockford, "JavaScript Object Notation (JSON)," July 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4627

[20] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107, 2008.

[21] K. Kovács, "Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase vs Membase vs Neo4j comparison," http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis, accessed March 15th, 2012, 2012.

[22] J.-J. Merelo-Guervós, A. Mora, C. Fernandes, and A. I. Esparcia, "Designing and testing a pool-based evolutionary algorithm," 2012, submitted to Natural Computing.

[23] R. Tanese, "Distributed genetic algorithms for function optimization," University of Michigan, Tech. Rep. CSE-TR-26-89, 1989.