Scaling in distributed evolutionary algorithms with persistent population

Juan J. Merelo-Guervós, Antonio Mora [*], J. Albert Cruz [†], Anna I. Esparcia-Alcázar [‡], Carlos Cotta [§]
[*] GeNeura Team
University of Granada (Spain)
Email: jmerelo@geneura.ugr.es
[†] Universidad de Ciencias Informáticas
La Habana, Cuba
Email: jalbert@uci.cu
[‡] Grupo S2
Email: aesparcia@s2grupo.es
[§] Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
Email: ccottap@lcc.uma.es

***Abstract*—This work presents the experimental results obtained with a distributed computing system created by mapping an evolutionary algorithm to the CouchDB object store. The framework decouples the population from the evolutionary algorithm and –through the API that CouchDB provides– allows the distributed and asynchronous operation of clients written in different programming languages. In this paper we present tests which prove that the novel algorithm design still performs as good as a canonical evolutionary algorithm and discover what are the main issues concerning it, what kind of speedups should we expect, and how all this affects the fundamental evolutionary algorithms concepts.**

***Index Terms*—I.2.m.c Evolutionary computing and genetic algorithms, C.1.4.a distributed architectures, H.2.4.d Distributed databases.**

I. Introduction

While algorithms have been traditionally designed with a single memory and CPU in mind, current technological infrastructure includes a high variety of frameworks and devices that twist and shift this paradigm in many different directions. Particularly for distributed evolutionary computation, the traditional notions of asynchronous, homogeneous and static computing systems have been superseded by others in which one or all of these features are absent [1], [2], [3], thus making the traditional distinction between master-slave and island-based models [4] moot by making them just two of all the possibilities that are created along the different feature axes.

Using these new foundations for evolutionary algorithms (EAs) allows to take full advantage of the performance of modern CPUs and operating systems and in some cases opens up the possibility of using new devices for distributed computing, making the participation in a distributed computation experiment as easy as visiting a website [5].

This change in the computing framework might –and usually does– imply changes in the algorithms themselves. A feature as usual as threads makes EAs escape the sequential cage, and make us rethink how the biologically inspired art of these algorithms [6] can be mapped to this new substrate. For instance, database management systems are nowadays a pervasive technology in business computing, but they were not used until 1999 as a base for persistent evolutionary algorithms by Bollini et al. [7]. They mention the fact that a database allows the simultaneous actuation of several clients, and change fundamentally the design of the EA from an *ab initio* strategy to an incremental one that makes use of the chromosomes that have been already created and evaluated, are stored, and can be efficiently retrieved from the database.

Database management systems have also changed and the last few years have seen the appearance of the so-called NoSQL, object or key-value stores [8]. Beyond the obvious fact that they do not use SQL for accessing data stores, these systems are characterized by being structured as key-value stores where the value is any kind of loosely structured document. In general, documents can include any data structure, although some of them (like, for instance, sets) might be present in only some of these systems. These generalized data structures can be described in languages such as XML or JSON (JavaScript Object Notation,[9]). The variety of languages for retrieving, aggregating and operating on data is as big as the number of different systems, but many of them have settled on JavaScript, due to the availability of efficient implementations and its widespread adoption by the web development community. Aggregating functions include in some cases *map/reduce* [10], [11], which is an efficient way of working on large amounts of data without needing large amounts of memory. Map/reduce requests are structured in a *map* function that is applied to every element within the selection, creating a couple of data structures that are *reduced* by performing some operation on them. A *map* operation might, e.g., create an array with the values of a certain field; a *reduce* operation will create a hash that records how many times each value appears.

These features are usually accessed through a REST (Representational State Transfer) API, a lightweight way of interacting with HTTP based servers which uses the semantics and syntax of this protocol. Since the only requirements for a language in order to create a wrapper around a REST API is to be able to make web requests and build strings, NoSQL databases can be accessed either easily from the command line, from the address bar of a browser, or from libraries built in many different programming languages. In either case, it does not add much overhead to the raw request: most of it lies in the conversion from the native NoSQL format (usually JSON) to the data structures in the native language.

All these features make NoSQL databases an ideal candidate for creating the backoffice for a distributed computation experiment; even more so when most systems allow replication, so that a multi-star (that is, multiple and linked servers with clients *hanging* from each one) infrastructure can be created and single points of failure avoided. In other papers about this system [12] we have presented its main features and proved that the concept works; we have also tested scalability and found which are the main bottlenecks that must be avoided to increase it.

In this paper we have refactored part of the system (proving once again that *implementation matters* [13]) that allows us to achieve higher speeds and better speed-ups, and our main focus is to explore the parameter space to find the best combinations. We will demonstrate that basic implementation parameters affect not only the speed, but also the behavior of the algorithm, having a definite influence in the number of evaluations needed to reach the optimum.

II. State of the Art

We will have to look at pool-based distributed EAs for the closest methods to the one used here. In these methods, several nodes or *islands* share a *pool* where the common information is written and read. To work against a single pool of solutions is an idea that has been considered almost from the beginning of research in distributed metaheuristics. Asynchronous Teams or A-Teams [14] were proposed in the early nineties as a cooperative scheme for autonomous agents. The basic idea is to create a work-flow on a set of solutions and apply several heuristic techniques to improve them, possibly including humans applying methods *by hand*. This technique is not constrained to EAs, since it can be applied to any population based technique (and from a wider perspective it lies at the heart of blackboard systems [15]), but in the context of EAs, it would mean creating different single-generation algorithms –with possibly several techniques– that would create a new generation from the existing pool.

The A-Team method does not rely on a single implementation, focusing on the algorithmic and data-flow aspects in the same way as the Meandre [16] system, which creates a data flow framework with its own language (called ZigZag), which can be applied, in particular, to EAs.

While algorithm design is extremely important, implementation issues always matter and some recent papers have concentrated on dealing with pool architectures in a single node environment: G. Roy et al. [17] propose a shared memory multi-threaded architecture, each thread having read access to the whole pool but write access to just a part of it. That way, interlock problems can be avoided and –taking advantage of the multiple thread-optimized architecture of today's processors– they can obtain very efficient, running time-wise, solutions with the added algorithmic advantage of working on a distributed environment. Although they do not publish scaling results, they discuss the trade-off of working with a pool whose size will have a bigger effect on performance than the population size on single-processor or distributed EAs. The same issues are considered by Bollini and Piastra in [7], who present a design pattern for persistent and distributed EAs; their emphasis is on persistence rather than performance, and hence they try to present several alternatives to decouple population storage from evolution itself (*traditional* evolutionary algorithms are applied directly on storage) and achieve that kind of persistence, for which they propose an object-oriented database management system accessed from a Java client. In this sense, our former take on browser-based evolutionary computation [5] is also similar, using for persistence a small database accessed through a web interface, but only for the purpose of interchanging individuals among the different nodes, not as storage for the whole population.

In fact, the efforts mentioned above have not had much continuity, probably due to the fact that there have been, until now, few (if any) publicly accessible online databases containing the pools. One of them uses the public FluidDB platform [18], combining evolution with (possible) stigmergy (communication through the environment); other than that, the pool based evolution is the same as the one proposed in this paper, since population is persistent, evolution is carried incrementally and the interaction among islands is only performed through the environment (in this case, the CouchDB DBMS). Our approach here differs however in that we consider a local database instead of a single copy of a global database accessed by all users, and thus latency problems can be avoided, hence relieving the scale problems presented by the authors of that paper (however, the results obtained in this work about packet size, and, in general, asynchronous organization of the algorithm can also be applied to the present work). Furthermore, this work advances the state of the art by introducing a novel fine-grained parallelization technique, and also by testing it on a distributed and real world environment.

III. SofEA, a CouchDB-based evolutionary algorithm

The first question is why choose CouchDB over other similar products, such as MongoDB or Redis. There have been several reasons for doing so: firstly, it is an open source product which is available in most Linux distribution repositories. This means that it is quite simple to create that infrastructure; besides, it uses JavaScript as its query language, which makes learning to use it very easy; besides, CouchDB introduces a new query language, UnQL[1], which is an hybrid between SQL and JSON. MongoDB can also be queried using JavaScript, but Redis does not, using instead its own language; thirdly, it uses persistent storage, which means stored procedures and data can be reused after reboot. Redis, on the other hand, handles everything in memory, which also implies a large RAM consumption. Finally, in the shape of DesktopCouch, CouchDB is a default install in most Linux desktops that include Gnome. This means that you can start to run this system *out of the box*, without needing additional installs in most cases (at least if you are using the preferred development system, a Unix or Linux box). Besides, CouchDB has as an advantage over MongoDB a greater adherence to web standards (the primary interface to the data is through RESTful HTTP) which grants it an advantage for use it in the Internet [19], and the possibility of using it in mobile platforms.

Mapping an EA to this system has to take into account its peculiar features and go with its grain to achieve maximum performance, both locally in the server and globally on the system composed of server+clients.

The first step is to decouple population from the rest of the EA, which usually include population as a variable that is passed around together with operators and the rest of the algorithm; even distributed EAs encapsulate the population and the rest of the algorithm in a single problem. In this case, and following Bellini et al. [7], we decouple population storage and its processing. Population will be stored in CouchDB. A *document* will include a chromosome, a random number and the fitness value. Besides, CouchDB includes two other pieces of data into each document: the key (which will coincide with the chromosome) and a version number.

This version number (or *revision*) will be used to characterize the state of a chromosome in the population:

- Revision 1: newly created chromosome, no fitness computed yet
- Revision 2: chromosome with fitness
- Revision 3: *dead* chromosome.

Revision numbers are updated natively by CouchDB; when a chromosome is updated with its fitness it is moved from revision 1 to 2; doing any further operation will take it to revision 3, but the only operation done by our system is to drop it from the *population*.

Since one of the strong points of CouchDB is its ability to stand a high number of simultaneous requests, the EA has been divided itself in four different programs, which will operate independently and asynchronously.

- Initialization: will create a set of chromosomes in revision 1.
- Evaluation: will take packets of chromosomes in revision 1, compute its fitness, upgrading them to revision 2 (in traditional EA parlance, they would be part of the *population*).
- Reproduction: packets of chromosomes in revision 2 will be crossed over and mutated; newly generated chromosomes are obviously in revision 1.
- Elimination: the population (chromosomes in revision 2) is culled down to a fixed number of chromosomes so that the less fit are progressively eliminated from it, taking them to revision 3.

[1]Specs available from http://www.unqlspec.org/display/UnQL/Home

These last three components are run at the same time, although they can be started asynchronously. In fact, they can be run in any sequence. Since the population is *out there* any part, or all of them, can be run in different languages, operating systems, processes or machines. This also allows to optimize the implementation of each one of them by using the language that suits them better. This *horizontal* division of labor has previously been used by Castillo et al. [20], in this case to take computational charge off a central server which is used mainly as a clearinghouse for distributing the population; the full GA, however, is run on one of the clients and there is a provision, in principle, for a single GA client with possibly several evaluators.

The main problem with our configuration is the *starving* of the algorithms, that is, the lack of chromosomes for performing its task. Since operation is asynchronous, if reproduction is not fast enough the evaluator will run out of chromosomes to evaluate; if evaluator is not fast enough, the reproducer will not have a sensible population to act on. The elimination phase is not so critical, but if it is not run frequently the reproductive population will grow out of proportion reducing the exploitative ability of the whole algorithm. This is fixed, in part, by making components wait one second if there is not enough material to act on; however, this increases the number of useless requests to the server. The main handle we can use to act on this is introducing a slight delay when starting them and changing packet size. However, ultimately the key is to have enough chromosomes to evaluate, since the reproduction phase can create new ones (maybe with less efficiency) even with a few.

One of the main advantages of this configuration is the fact that every chromosome is evaluated just once. Since we use the chromosome string as a key, if the reproduction attempts to reintroduce a chromosome it will return an *already existing* error. This means that the reproduction phase (and, in fact, also the evaluation phase) becomes increasingly less successful with the ongoing algorithm, but also that every individual in *revision 2* is unique and thus (genotypic) diversity is mostly kept, no matter what kind of EA we use.

Since several evaluators and reproducers can act concurrently, we should issue them different chromosomes to work on. One of the possibilities would be to keep tabs on the server of the last one issued, but this is a problem since there is no guarantee that the result will be returned, and then it would also cause starvation if a slow client takes the last chromosomes to evaluate. So we included the random number in the document, which is used to sort the population and retrieve all the chromosomes whose random constant is higher than that first random number. There are two problems with this: if this number is too high, less chromosomes than the established packet size will be returned. This could be avoided by issuing another request for the remaining number of chromosomes, however this is cumbersome and it is not really a big problem to have less chromosomes to operate on; the second one is that there is small probability (which increases with decreasing population size) that the same chromosome is returned twice to two different clients. This is not a problem in the reproduction phase, but it could be during evaluation, causing a conflict, as we will later see.

One of the problems with this system is to make all clients know the algorithm has finished, and make it so as soon as possible. In general, the approach is to make the client that finds the termination condition create (actually, update) a special document that is periodically consulted by all other clients. If the termination condition is to run a certain number of evaluations, it is the *reaper* who creates it; in the experiments described in this paper, as soon as an evaluator finds the chromosome with all ones, it creates the document and stops running. After every iteration in their loops, all other clients check this document, and when it indicates the experiment is finished, they stop. Since this involves a single retrieval of a document with a known key, the impact on client performance is very small.

Clients have been written in Perl and JavaScript (which has been used also for writing CouchDB views) and are available with a GPL licence from https://launchpad.net/sofea.

IV. Experiments and Results

In this section we will first test the influence of the implementation parameters on the result, and then try to measure the speed-up obtained when multiple clients are added.

A. Testing the influence of implementation parameters in the algorithm

As mentioned above in the introduction, we will check the influence of the parameters in the results. We will be using the classical OneMax example, which reaches the optimum when all bits in a string are equal to 1. Chromosome length is 128 and in all cases an initial population of 128 was generated before starting to run the algorithm. The clients were run until the optimal solution was found, and we changed two parameters: the *base population* (the number of *live* individuals the *reaper* leaves every time it acts) and the evaluator and reproducer population (the number of individuals every one of them receives –and, in the case of the reproducer, also outputs– every cycle). The combinations and their denominations are shown in Table I.

TABLE I
Combination of parameters and denominations we are using for it.

Denomination	Base Population	Evaluator packet size	Reproducer packet size
p128-e32-r32	128	32	32
p128-e64-r32	128	64	32
p128-e64-r64	128	64	64
p64-e32-r32	64	32	32

Instead of exhausting all possible parameter combinations and packet sizes, we have used a reasonable set of combinations; we have excluded sizes smaller than 32 since they would increase the number of requests; the packet size for the reproducer is never bigger than for the evaluator, since that would create too many individuals the reproducer would be unable to evaluate. Both packet sizes are smaller than the base population, which gives each generated individual a chance toreproduce before being eliminated. In general, this combination has been found to reduce waiting time in every client (due to lack of suitable genetic material) and makes the algorithm to run smoothly, without making the number of non-evaluated individuals increase excessively until the end of the algorithm. Every configuration is repeated ten times.

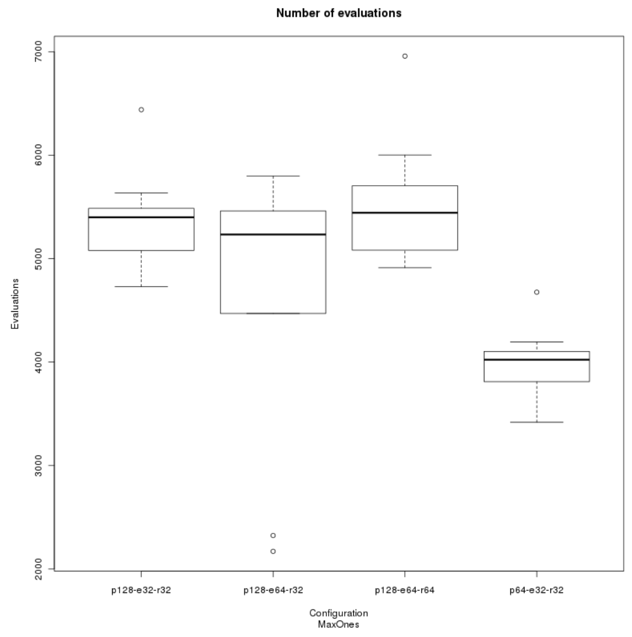

Fig. 1. Boxplot of the number of evaluations for different combinations of parameters; the meaning of the labels in the x axis is shown in Table I. Differences are significant among p64 and the rest, but not among these.

The time and number of evaluations to success (AES, average evaluations to success) are plotted in Figures 1 and 2. Focusing first on the number of evaluations, although we find small differences depending on the packet size which indicate that it might be better to make the packet size for the reproducer smaller than for the evaluator, the main factor is the base population size. While for population = 64 the number of evaluations is on average 3,975, when the population doubles the average hovers around 5,000, although the smallest value corresponds to evaluator packet size = 64 and reproducer packet size = 32, with an average of 4,627. In this sense, results are in accordance with what is usual in genetic algorithms, with smaller populations generating less useless individuals and thus being able to find the solution quicker.

However, the second Figure 2 shows a different scenario. Since the number of evaluations is smaller, again the smaller population beats the other time-wise; but the biggest factor in

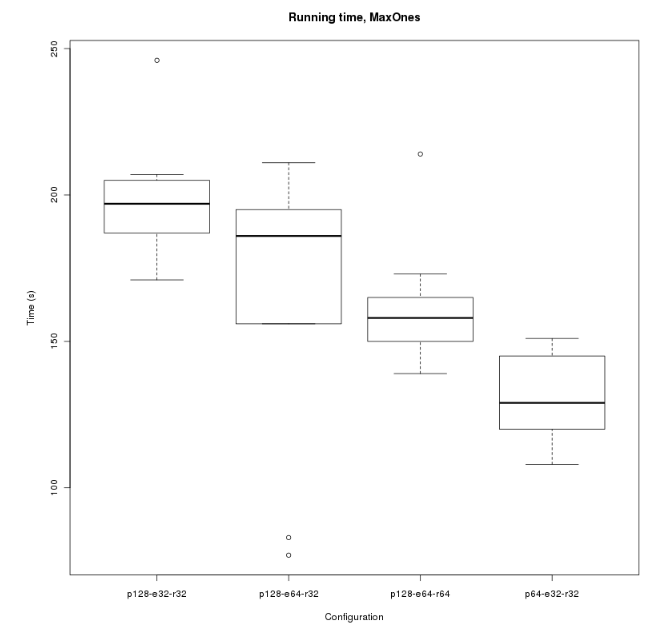

Fig. 2. Boxplot of the time in seconds needed to find the solution for different combinations of parameters; the meaning of the labels in the x axis is shown in Table I. Differences are only significant among the first and the rest.

the number of evaluations performed per second seems to be the packet size. When both evaluator and reproducer packets are equal to 64, it needs on average 161.3 seconds to find the solution; the other need on average more, almost 200 seconds if the packet size is halved. In general, that difference can be explained by the number of requests needed to finish the algorithm. Taking into account that the number of evaluations is roughly the same, p128-e64-r64 will need $E+R$ requests, p128-e64-r32 will use $E+2R$ and p128-e32-r32 $2(E+R)$.

This gives us a rule of thumb to run the algorithm to success: try to make the population as small as possible, packet size for clients which is equal to half the base population size; however, this rule is for a baseline configuration and will have, later on, to check it to see whether it scales up properly. That way, the best combination of AES and running time will be achieved. This will be tested in the next experiment, where we will try to check the speed-up behavior by multiplying the number of clients.

B. Speed-up behavior when adding clients

This experiment will be run as follows: keeping the *reaper* to a single client, the number of evaluators and reproducers will be changed. The dynamics of evaluation and reproduction is complex and the number need not be the same: since they act asynchronously, adding more clients run the risk of running into the conflict of several evaluators trying to evaluate the same set of individuals; or else, both clients can *starve* if they do not have enough individuals to act on. Besides, the parameter space is huge and we cannot but explore a small part of it: even if we keep the base population constant, the block size for both clients and the number of clients of each one can be changed in many different combinations.

To try and reduce this possible number of combinations, we have run a profiler (which is a good practice in the design and development of evolutionary algorithms [13]) to identify beforehand which can be the best block size. We know that block size does not affect much the quality of the algorithm but it affects running time via the response time of the database; besides, since response time is based in the size of the request and response but also on the operations run on the set, it might not be exactly linear with size. At any rate, by running a Perl profiler (`Devel::NYTProf`) on the client we have found that client running time is dominated by the interaction with the database and –as we have supposed in advance–, there is a big influence in block size. The critical request is the instruction that requests a block of individuals in the state we need them (Revision 1, without evaluation or Revision 2, ready for reproduction), which can take from 50 ms. (smallest size) to 200 ms. (bigger size). We thus chose 16 as universal block size, eliminating one parameter. Besides, profiling allows us to optimize map/reduce functions running within CouchDB to make them as fast as possible and eliminate unneeded requests that were used mainly for statistical purposes.

Once the block size was fixed to 16, we tried different client configuration, varying the number of evaluators and reproducers as shown in Table II.

TABLE II
COMBINATION OF NUMBER OF EVALUATORS/REPRODUCERS AND DENOMINATION WE ARE USING FOR IT.

Denomination	Evaluator #	Reproducer #
E1R1	1	1
E2R1	2	1
E4R1	4	1
E6R2	6	2
E8R2	8	2

Each experiment was repeated ten times with the same initial conditions, that is, a clean database just created; before each iteration, all individuals in the database were erased twice since, due to the asynchronous termination of clients, sometimes it took some time for one of them to finish. At the end of each run the database occupied around 1-2 Gigabytes which can result in some degradation of performance; that is why it was started anew each time. All clients were run in the same computer: an AMD six core with 24 GB memory running Ubuntu 10.10. CPU occupation was monitored and even in the case of a dozen clients running at the same time, no excessive CPU load was observed, so that if there is any degradation in running time it must be due exclusively to the program itself, not to the operating system context.

The first thing we are interested in is, of course, whether there is any improvement with the number of clients. Running time in seconds and using nomenclature shown in Table II is shown in Figure 3. The figure shows that, for these configurations, adding clients result in a definite improvement in running time. The most dramatic time is when a single evaluator is added to the base configuration; average running time

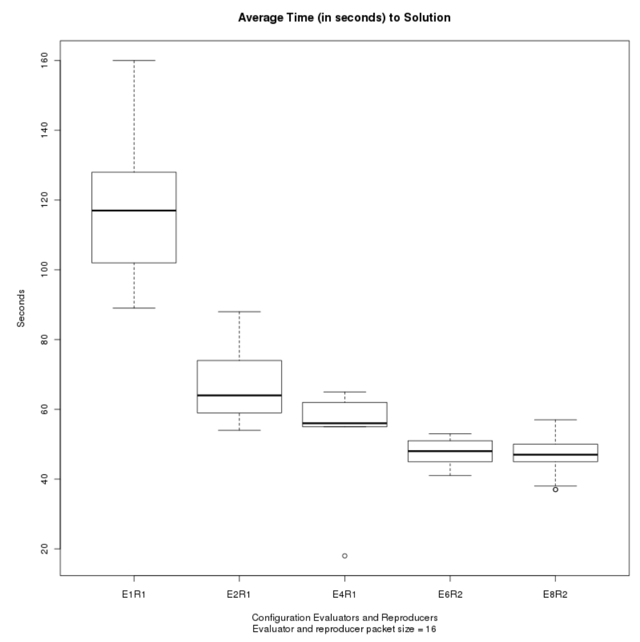

Fig. 3. Boxplot of running time (in seconds) for different client configuration.

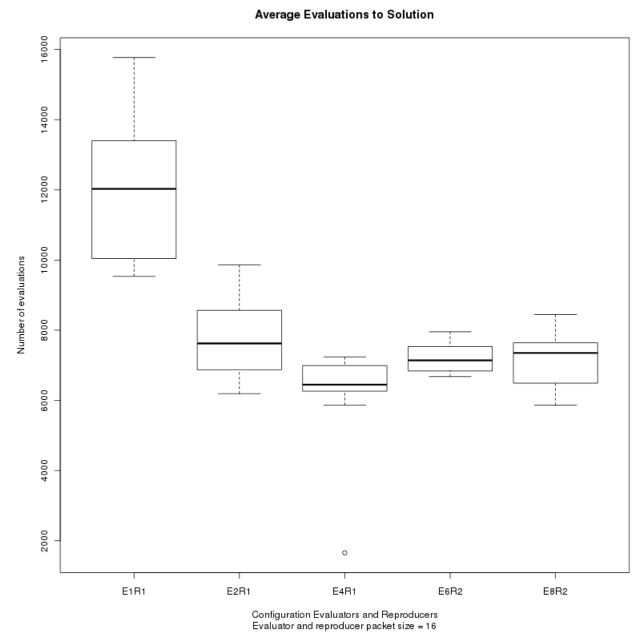

Fig. 4. Boxplot of the number of evaluations to solution for the different number of clients, denomination is the same as in Figure 3.

goes from 118.0 down to 67.4 s. Decrease is not so dramatic for the rest of the configurations, but it does reach a minimum average of 46.95 when 8 evaluation and 2 reproduction clients are used; this amount is almost 1/3 of the initial running time. It is difficult to estimate the actual speed-up, since computing nodes are dissimilar but it is clear that there is an improvement in running time for these configurations. We also tested a configuration with 9 evaluation and 3 reproduction clients, but there was no further improvement and even a slight increase in running time; this was to be expected, since speed-up cannot increase to infinity. However, since the problem is not in the central server, but probably in the fact that the number of *simultaneous* evaluations, 128, is the same as the base population, we should expect further increments when population size increases; we should also look for parallelization strategies that avoid these problems.

The second issue we will study is the cause of this improvement; an obvious first factor will be raw computing power, but, as mentioned above, we are dealing with a complex dynamic system which is bound to have influence in the algorithm dynamics, resulting in different number of evaluations to solution. Results are sown in Figure 4, which roughly show the same pattern as Figure 3 with a decrease in the number of evaluations up to E4R1 but an *increase* for E6R2 and E8R2. The decrease, anyways, is less dramatic than running time, which implies that parallel execution explain part of the improvement in running time for the first 3 configurations, and all of it for the two later experiments. Even as the algorithm is doing *more* evaluations, some of them are being executed in parallel, resulting in *less* time. From this we can conclude that, in general, SofEA runs efficiently in parallel environments and could be used profitably with several, possibly distributed, clients to improve running time of a sequential evolutionary algorithm. This option becomes increasingly interesting when evaluation time becomes bigger in relation to communication time.

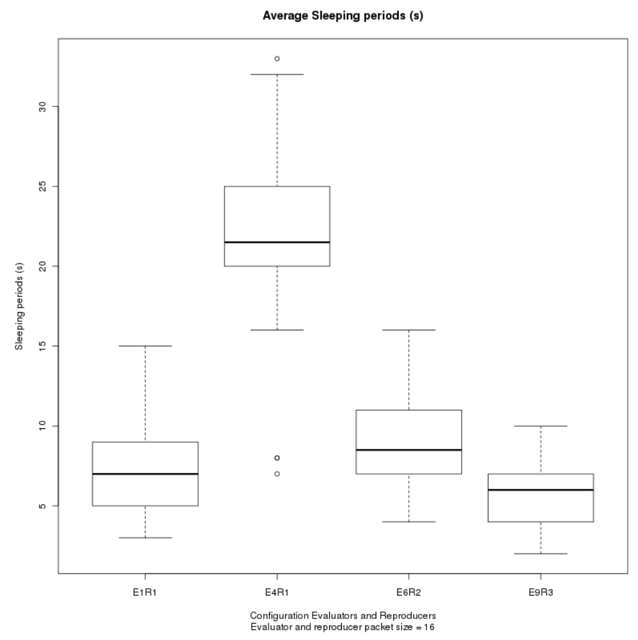

Fig. 5. Boxplot of number of sleeping periods for different client configuration.

However, it is also interesting to know why these running times are achieved and we will look at two of the factors that cause the clients to perform less that would be expected of them. The first one is the number of periods in which

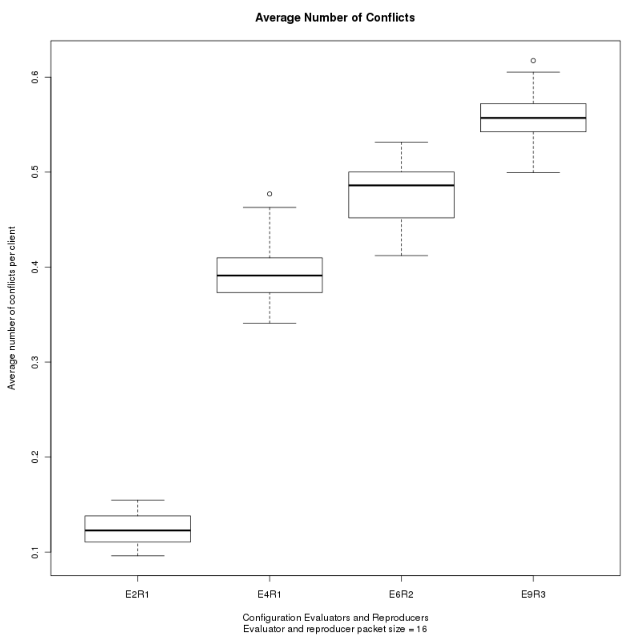

Fig. 6. Boxplot of the number of conflicts, that is, attempts to update an individual that has been already evaluated (by another client). Denomination is the same as in Figure 3.

the clients *sleep* due to lack of chromosomes. Figure 5 shows for four different (and representative) configurations the number of *sleeps* the clients incur when there is absolutely no chromosome in a state they can work with. In this case it is plotted for evaluators which sleep when there is no new chromosome. As we can see, there is an increase from the single-couple-of-clients scenario E1R1 to the case in which there are 4 clients. Comparing this figure with Figure 3, that shows that average running time is around 60 seconds, we can see that, out of that time, on average about 20 seconds are spent *sleeping*; if we could reduce that wasted time the running time of the whole system could be increased even more; it is not a simple task, however, since increasing the initial amount of chromosomes would imply dealing initially with too many random chromosomes and increasing the number of chromosomes created by the reproducer each step would also increase the time it takes to process them, increasing the risk of *starvation* which leads to increased *sleep* time. However, by looking at the right hand side of the figure, which increases the number of reproducers, we can see that the evaluators are kept working, even if there is triple number of evaluators than of reproducers, to the point that sleeping time is less than 10 seconds; however, taking into account that they run for a shorter time, the percentage of time kept sleeping is bigger than in the cases with a single reproduction client.

Be as it may, these configurations have other problems, mainly related to the fact that its throughput is much higher, so they run the risk of having two clients simultaneously processing the same individual. The number of evaluator conflicts are shown in Figure as a rate, that is, number of already evaluated individuals over the total number of evaluated individuals. For obvious reasons E1R1 is not shown in this graph, but we find an increase of conflicts as we increase the number of clients or –expressed in another way– as the number of individuals covered approaches or exceeds the base population, In the case of E9R3, when the number of individuals processed *simultaneously* is equal to the base population, the number of conflicts can be up to 60%. This is, in part, due to a design decision: every individual includes a random number and CouchDB returns *block-size* individuals from a generated random number up. When population size is relatively big, there is very little chance that the individuals returned are the same (but it is not negligible, as shown in the E2R1 case). However, that probability increases with the number of clients, to the point that when the clients additively process a high percentage of the base population overlapping is almost assured.

Despite these drawbacks, the fact that speed-up is achieved anyways shows that, in principle, SofEA could be used for distributed evolutionary computation experiments with different number of clients, as soon as these number of clients is set up following a few rules of thumb, such as:

- There should be more evaluators than reproducers; of course, this will depend on the difference of speed between evaluation and reproduction but, in general, reproduction will be faster.
- There should never be more simultaneous evaluations than the base population.
- The block size for evaluators and reproducers should be kept as small as reasonably possible.

At any rate, we have proved that an efficient and scalable evolutionary algorithm can be designed and implemented using CouchDB as its foundation.

V. Conclusions, discussion and Future Work

In this paper we have shown how SofEA, a distributed evolutionary algorithm that uses as back-end the CouchDB object store, can be used to create a framework for distributed scalable systems. The system is asynchronous and is parallelized at the algorithm level, putting evaluation and reproduction in different computing nodes. We have explored the client configuration space showing that there are certain configurations that allow for a better speed-up; speed-ups of up a to a third of the *sequential* (actually, single pair) configuration can be achieved in simple problems.

We have also examined the hurdles for even better speed up: clients falling out of lockstep so that one of them does not produce enough individuals for the others to work on and making it going to *sleep* and overlapping of the chromosomes that are going to be evaluated producing conflicts and wasting the time spent by one of the clients evaluating them. However, these situations are predictable in a way and can be overcome by judiciously setting up the number of clients following a few rules of thumb.

Since both these obstacles are the result of design decisions, it could be argued that other options could be examined. For instance, the simple fact of decreasing the time the client goes to sleep could improve this. This can be examined, of course, but one possible outcome is that the number of *periods* could be increased. In fact, we tested one configuration in which the second evaluator started one second after the first, to avoid them working on the same population. However, this did not lead to any improvement, results being statistically undistinguishable. A possibly better option could be to change dynamically the block size; if less than the initial block size are available, this is a signal that the supply is going down and might eventually halt. The problem with this is that, even if the sleep time is decreased, the number of requests are increased, leading to the same or more running time. However, this is a venue that might be interesting to pursue.

The source of evaluation conflicts, on the other hand, comes from the fact that we do not keep tabs of which individuals in the database have been sent. Doing so would imply first sorting, which has to be done via a (slow) map function since CouchDB uses a single key and we already use it for avoiding repetitions of an individual, and also having a stored *cursor* keeping tabs of which one has been sent; but that would also imply using some timeout mechanism just in case unreliable clients do not return evaluation, so this method for the time being is discarded. Another option would be to just use bigger populations, but that slows down the algorithm and also the operation of the database, so it is a path that we will try to avoid for the time being.

The experiments done here open a good amount of possibilities. The client type is decided beforehand; since the clients are served from the database, some intelligence could be added to it so that it is able to decide which clients are needed the most, even during the execution of the algorithm. If too many non-evaluated chromosomes are present, an evaluator can be served; else, a reproducer. The type of the client can even be changed in running time, and its parametrization too. Even different algorithms could be run in each one of them.

It would be interesting to test also the system with more heavy-duty problems, such as MMDP or p-Peaks. These imply a higher number of evaluations, but also a fitness function that takes longer to evaluate. Since it also needs a bigger population (in the canonical GA case, at least), we might overcome some of the hurdles found in this paper and achieve better speedups.

There is also some room for optimization of the CouchDB server by reducing the number of heavy-duty requests. Eventually, we expect to achieve speeds for the single clients system that are competitive with those achieved by a sequential system. Other features of the system, such as the `_changes` feed (a stream of all changes made to the database), could be used to make the algorithm more reactive to changes in the population, since this feed contains all changes made to the database; this would also include the creation of clients using `node.js` or other event-based systems.

Acknowledgments

This work is supported by projects NEMESIS (TIN2008-05941) and TIN2011-28627-C04-02 and TIN2011-28627-C04-01,awarded by the Spanish Ministry of Science and Innovation and P08-TIC-03903 awarded by the Andalusian Regional Government.

References

[1] J. Atienza, M. García, J. González, and J.-J. Merelo-Guervós, "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, 2000, pp. 1087–1089, http://citeseer.nj.nec.com/atienza00jinetic.html.

[2] Martina Gorges-Schleuter, "ASPARAGOS: An asynchronous parallel genetic optimization strategy," in *Proceedings of the Third International Conference on Genetic Algorithms*, J. D. Schaffer, Ed. Morgan Kaufmann Publishers, 1989.

[3] B. Bánhelyi, M. Biazzini, A. Montresor, and M. Jelasity, "Peer-to-peer optimization in large unreliable networks with branch-and-bound and particle swarms," *Applications of Evolutionary Computing*, pp. 87–92, 2009.

[4] M. Nowostawski and R. Poli, "Parallel genetic algorithm taxonomy," in *Knowledge-Based Intelligent Information Engineering Systems, 1999. Third International Conference*. IEEE, 1999, pp. 88–92.

[5] 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

[6] D. E. Goldberg, "Zen and the art of genetic algorithms," in *ICGA95*, J. D. Schaffer, Ed., George Mason University. San Mateo, California: Morgan Kaufmann, June 4-7 1989, pp. 80–85.

[7] 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.

[8] D. Bartholomew, "SQL vs. NoSQL," *Linux Journal*, vol. 2010, no. 195, p. 4, 2010.

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

[10] H. Yang, A. Dasdan, R. Hsiao, and D. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 1029–1040.

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

[12] J. J. Merelo-Guervós, A. Esparcia-Alcázar, A. M. Mora, and J.-A. Cruz, "Pool-based distributed evolutionary algorithms using an object database," in *Proceedings EvoApps 12, to be published*, 2012.

[13] J.-J. Merelo-Guervós, G. Romero, M. García-Arenas, P. A. Castillo, A.-M. Mora, and J.-L. Jiménez-Laredo, "Implementation matters: Programming best practices for evolutionary algorithms," in *IWANN (2)*, ser. Lecture Notes in Computer Science, J. Cabestany, I. Rojas, and G. J. Caparrós, Eds., vol. 6692. Springer, 2011, pp. 333–340.

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

[15] B. Hayes-Roth, "A blackboard architecture for control," *Artificial Intelligence*, vol. 26, no. 3, pp. 251–321, 1985.

[16] X. Llorà, B. Ács, L. Auvil, B. Capitanu, M. Welge, and D. Goldberg, "Meandre: Semantic-driven data-intensive flows in the clouds," Illinois Genetic Algorithms Laboratory, Tech. Rep. 2008103, 2008.

[17] 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.

[18] J. J. Merelo, "Fluid evolutionary algorithms," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.

[19] S. Tiwari, *Professional NoSQL*. John Wiley & Sons, Inc., 2011.

[20] P. A. Castillo, M. García-Arenas, A.-M. Mora, J. L. Jiménez-Laredo, G. Romero, V. M. Rivas, and J.-J. Merelo-Guervós, "Distributed Evolutionary Computation using REST," *CoRR*, vol. abs/1105.4971, 2011.