SofEA, a Pool-Based Framework for Evolutionary Algorithms using CouchDB

Juan J. Merelo, Antonio M. Mora, Carlos M. Fernandes
University of Granada
Department of Computer Architecture and Technology, ETSIIT
18071 - Granada
jmerelo,amorag,cfernandes@geneura.ugr.es

Anna I. Esparcia-Alcázar
S2 Grupo
aesparcia@s2grupo.es

ABSTRACT

This paper studies SofEA, an architecture for distributing evolutionary algorithms (EAs) across computer networks in an asynchronous and decentralized way. SofEA is based on a pool architecture which is implemented using an object store interacting asynchronously with several clients. The fact that each client is autonomous leads to a complex behavior that will be examined in this paper, so that the design can be validated, rules of thumb can be extracted and the limits of scalability found. We will show how, beyond the usual measures employed in EA, specific measures such as the number of conflicts across clients can give us hints on the algorithm behavior, and how implementation details can change not only the running time, but also the behavior of the evolutionary algorithm itself. By using these measures we try to find ideal values for parameters such as the simultaneous number of individuals evaluated by a client or the way these are chosen from the pool.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; G.1.6 [**Mathematics of Computing**]: NUMERICAL ANALYSIS—*Optimization*; D.2.8 [**Software Engineering**]: [Metrics complexity measures, performance measures]

Keywords

Cloud Computing, Cloud Storage, Evolutionary Algorithms, Distributed Algorithms, NoSQL databases, key-value stores, complex systems

1. INTRODUCTION

Modern computer systems offer the promise of massive scalability, fault tolerance and self-adaptiveness if only the algorithm can be properly adapted to it. However, *traditional* parallel systems pose several problems that hinder their scalability.

Master/slave models, for instance, rely on a single server which is a single-point-of-failure, and are limited by its capacity and bandwidth. The fact that all clients usually act synchronously also makes the master/slave more suitable for installations with similar, or the same, power and limit its adoption for massive, ad-hoc distributed evolutionary algorithms. However, since they have been used for systems such as SETI@home [**?**] indicate that the number of clients can escalate massively, at least if the clients act in a publish-subscribe model (with the server sending, not being polled, by the clients) and do not require much computation to serve or create requests. This is not always the case in distributed evolutionary algorithms: if the EA runs on the server, and evaluation is farmed out to clients (as was done, for instance, in [**?**, **?**]), scalability is limited; the situation improves if the evolutionary algorithm is run on the clients [**?**], using the server just as a clearinghouse for interchange of immigrants.

However, even with the introduction of a centralized server, this kind of layout rather corresponds to an *island model* [**?**] where each client is an *island*. *A priori*, there are no limits to scalability, since using either a central server for interchange of information or creating an ad-hoc network any amount of islands can participate in an experiment. However, connectivity has limits, bandwidth is not free, and in practice, fixed topologies are used, which restricts the amount of islands that can be added and prohibits spontaneous incorporation of another node. In practice, too, synchronized experiments are often used, requiring homogeneous networks, and self-adaptability is not used. A special kind of *island models* are the so-called *cellular* evolutionary algorithms [**?**], which are basically single-individual nodes with limited connectivity to other islands (often only to those around them in a plane either in a hexagonal or square layout).

These cellular algorithms are well adapted to some special problems, but pose the same drawbacks as above: synchronicity and rigid connections. However, since connections are limited, they have often reached a good speed-up. The best architectures in this sense are P2P systems such as EvAg [**?**], with an emerging connectivity and no centrally issued command; they offer the scalability of cellular genetic algorithms with excellent fault-tolerance [**?**]. There is no problem indeed with the algorithm *per se*, except that, being an ad-hoc software system, its adoption is limited and does not make use of the infrastructure already available.

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.

At any rate, EvAg is an excellent system and we only aim to provide in this paper another alternative for those who are already acquainted with the tools used in it.

The system described in this paper, SofEA, is a pool based distributed evolutionary algorithm. It uses a central or replicated pool for storing the population, decoupling the methods applied to it from its storage. The pool is stored in CouchDB, a key/store database management system [**?**], which in turn leads to a series of design decisions when implementing the evolutionary algorithm (implementation *always* matters [**?**] if the maximum, or even an acceptable, performance is to be reached). Our objectives with the design of this system is to create a distributed evolutionary architecture that is able to adapt to spontaneous clients joining, work asynchronously and achieve good speed-ups without requiring special-purpose hardware. In this paper we will show the design choices available to us and validate them through experimental measures on the system itself, using different configurations.

The rest of the paper is organized as follows: Next section presents the state of the art in pool-based algorithms and its particular instantiation for evolutionary algorithms. Our own version is explained in section **??**, which is tested next in the experimental section (**??**). Finally, we draw some conclusions and propose future lines of work in section **??**.

2. STATE OF THE ART

In their interesting study of the state of the art in asynchronous evolutionary algorithms, Alba et al. [**?**] note that the model should be distinguished the implementation, although, as indicated in the introduction to this paper, there is a feedback between them in such a way that they must fit each other. However, we will first look at models, and then at implementations.

The most popular model of asynchronous distributed algorithms is called *A-teams*, where A stands for *asynchronous* [**?**, **?**, **?**]. A-Teams combine different algorithms in closed loops that share a memory and are a way of specifying data flow among different methods to solve a problem. A-Teams are not evolutionary methods but have been successfully applied in the last decades to a wide variety of problems [**?**]; their authors have released a toolkit that can be used to implement solutions to different problems. It is interesting to note that A-Teams include *constructors* as well as *destructors* distributing concerns among different agents which cooperate to build or ultimately find the solution. Since their emphasis is in dataflow, they are similar to the more up-to-date system *Meandre* [**?**], a cloud-based dataflow architecture which is, in particular, used for evolutionary algorithms [**?**] using a MapReduce framework, that is, a system in which massive amounts of data are first mapped by applying a function to them and then the result *reduced* by applying a function to the resulting data structure. In this case, data in the form of a population of individuals that undergo several transformations along the data pipeline is mapped to individual computing nodes and then reduced to proceed to the next stage. In that sense, lately there are many papers [**?**] that describe evolutionary algorithms that use MapReduce frameworks. All these systems are not so amenable to asynchronous and spontaneous collaboration, though.

These models can be implemented in many different ways, but they often refer to a *pool* from which solutions can be drawn, improved and put back, or to where newly constructed solutions can be shared among all agents participating in the experiment. Taking then one step down, several authors have directly implemented evolutionary algorithms in a pool based architecture, where the basic idea is to use a store of solutions from which the evolutionary algorithm draws its individuals, instead of 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 [**?**]; lately, multi-threaded systems with a shared memory [**?**] from which all can read, but is writable only by one of the threads, and even relational database systems [**?**] have been used, acknowledging their capability for avoiding explicit synchronization and its fault-tolerance, at least to client failure, providing a persistent storage for population from which solutions can be, later on, retrieved.

SofEA is an attempt to design a pool-based evolutionary system that is asynchronous, fault-tolerant and highly scalable. In this paper we examine the trade-offs that lead to the different decisions we have made, and show the system capabilities. The scaling behavior is not the focus of this paper, although in other publications [**?**] we have proved its limits and made suggestions on how these can be overcome.

3. SOFEA, A COUCHDB-BASED EVOLUTIONARY ALGORITHM

The reasons for choosing CouchDB over other similar offerings have been explained elsewhere [**?**], but they boil down to the presence of all the features we needed (MapReduce functions, availability of client libraries in several languages, and good performance). However, using other similar NoSQL systems will be researched in the future, although we would like to emphasize that object data stores do not have a standard API and we would probably need to adapt the model to each one's peculiarities.

CouchDB is an key-value store [**?**] that uses JSON (JavaScript Object Notation, a text serialization of arbitrary data structures [**?**]) as its *lingua franca*, being able to store any kind of data structure. Objects can be retrieved by key or range of keys directly, but complex queries using MapReduce [**?**] 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* are applied to lists of them organized in key/value pairs. CouchDB uses a simple REST (Representational State Transfer) API that can be accessed from the command line or multitude of client libraries; this API can be used either to access objects directly or to apply operations on them. Every object 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; its main part changes by one every update.

Mapping an EA to this system has to take into account its peculiar features and use them to achieve maximum performance, both locally in the server and globally on the system composed of server and clients; when doing this mapping, we will have to bear in mind the objectives we put forward at the beginning: good scalability and leveraging of CPU power, client autonomy and fault-tolerance.

There are in principle many different ways of implementing a pool-based evolutionary algorithm over CouchDB. The first choice was to store directly population on it, using

clients only for computation so that the setup is more alike a *master/slave* model using farming for computation, except that evaluation as well as the evolutionary algorithm itself will be done by the clients. Chromosomes will be then objects, and to ease retrieval, we chose to use the chromosome itself as key, with the value including the chromosome string, fitness, and a random constant we will use as part of a primitive partitioning method. Since the chromosome string is the key, every chromosome will be present only once in the pool, avoiding repetitions and thus naturally increasing diversity.

Revisions are a straightforward way to represent the state of the chromosome. When it is created, its revision is set to 1. When it is evaluated and fitness added, revision turns to 2; it is just natural to use revision 3 as a "dead" state. Revisions will be, then, used to select chromosomes in one or other state.

The *selection algorithm* is an integral part of the evolutionary algorithm. Most selection algorithms work on a population, or assume that selecting several random chromosomes is an easy operation. In our case, selection will operate on chromosomes whose fitness has been computed *revision 2*, but selecting random chromosomes from it is not a trivial operation a priori. However, we will use a view that returns chromosomes ordered by the random number they include to select just a pre-established amount of them, and at the same time avoid (although not completely) overlap of selected chromosomes. At the same time, selection will have to eliminate the worst chromosomes so that the evolutionary algorithm is effective.

Elimination of the worst chromosomes will be done on the first of the several clients that will be used. Since the population is decoupled from the algorithm, we can also decouple selection from the rest so that it operates autonomously, cutting down the population in revision 2 (with fitness) to a manageable value. This client is called the *reaper*, and for a base population of p, and a r chromosomes in state 2, *eliminates* (takes to revision 3).

The rest of the clients can also be independent; there is no need to loop over selection, reproduction and evaluation; selection has been already detached, and we will do the same. Detaching reproduction and evaluation has its advantages: they can be written in different languages looking for the most efficient implementation or run in different machines adapted to its purpose. Efficient modern architectures allow to run both processes (or both threads) at the same time without a noticeable effect on the rest of the tasks running, so that, in effect, they can operate as a pipeline, with one process (or, obviously, set of processes) applying evolutionary operators while others compute the fitness of previously generated individuals (and yet another, as seen above, limits *living* population). In principle, they will both operate on a number of individuals selected randomly (as seen above) from the population. To kick-start each experiment, all existing documents are deleted and a initial random population is created so that the evaluator(s) can start working immediately, with the reproducer(s) kicking in a bit after that.

To make all clients know when the experiment has finished, a document with a known key is used; this key can be "solution" when we want to find a specific solution, or "evaluations" when we want to stop after a number of evaluations have been made. This document is stored in the same CouchDB as the rest of the chromosomes; since CouchDB is schema-less, the same database can be used to store all kinds of objects. In fall, all program logic is either stored as views in CouchDB or in the clients, which is, for us, a good reason to use CouchDB instead of other similar systems.

That division among clients meets most of the requirements for our system. A client only needs to know the database address to start adding its capacity to the evolutionary algorithm; if it stops working, the rest of the clients will continue making the system fault-tolerant if at least one reproducer, evaluator and reaper are present (and, obviously, the server). Operation of each client is decentralized except for the termination condition, that will be checked when each request is serviced. It can use efficiently available resources since memory and CPU consumption are small, with dozens of clients operating at the same time, even in a single machine. And, finally, we can achieve some speedup by adding clients, although a trade-off between the number of simultaneous requests and the number of individuals processed in each one must be found; it does not make much sense if we make a request for every single evaluation, for instance, since servicing it will consume much more time than doing the evaluation.

In general, then, the model we use is a client-driven (not data-driven) evolutionary pipeline that is actually implemented using limited-size individual blocks and clients in different processes. The model does have some problems, first and foremost the existence of several variables (block size, number of clients, relation among block sizes for the different clients), but also these problems, which are inherent in the design:

- Oversupply of chromosomes in revision 1 (non-evaluated). If the evaluator lags behind just a bit, the reproducers will create too many chromosomes that will be left without evaluation.

- Evaluator starvation due to too few chromosomes to evaluate. The reproducer will always have enough to work on, except maybe at the beginning of the experiment, since population is never completely eliminated, just reduced.

- Conflicts: since the block of individuals is selected randomly, those selected by two clients can overlap. This will result in a lack of efficiency for the evaluator, and loss of diversity for the reproducer; the first is unavoidable, but its effect is probably smaller than keeping tabs and blocking chromosomes already sent to a client; and the second will affect mainly the evolutionary algorithm, but not greatly, since anyways two copies of the chromosome cannot be present in the population.

The main task of this paper will be to validate these choices by evaluating SofEA on a simple problem, OneMax. Since our objective was mainly to get a grasp of the complexities involved in designing an algorithm adapted to this platform, we think that the choice of problem is not important, being the main factor the time needed to compute the fitness function; eventually we will apply SofEA to other problems and see whether conclusions drawn from this simple problem can be extended to them. We will also study different implementation alternatives by measuring running

times (for implementation efficiency) and number of evaluations (average evaluations to solution), to contrast the effect of implementation choices on the algorithm itself.

For this paper, the clients have been written in Perl and JavaScript (which has been used also for writing CouchDB views) and are available with a GPL license from `https://launchpad.net/sofea`.

4. EXAMINING THE BEHAVIOR OF SOFEA

Since clients operate autonomously and asynchronously, there are parameters at the model and algorithm levels, and they feed back on each other, some tentative exploratory analysis will have to be made; first to compute the values for parameters that yield the best performance, examine the robustness of the algorithm and finally find out the most significant measure needed to explain the behavior obtained. All experimental parameters are as shown in table **??** unless told otherwise. Experiments were made in a single computer, an AMD six-core running Ubuntu 11.04 with 16 GBytes of memory and a SSD disk drive hosting the database. Experiment data and parameters are also available from the URL mentioned above.

Table 1: Common experiment parameters.

Parameter	Value
Repetitions	10
Chromosome size	128
Initial population	128
Termination condition	Solution found

4.1 Initial population

Initial population is an implementation parameter; an initial set of random chromosomes must be supplied so that the evaluator can start working immediately. Too many of them will mean too much exploration, while too few of them will starve the evaluator very soon if the reproducer cannot keep up with it. We have tested two different quantities: 128 and 256, for a fixed evaluator block size of 128. This parameter should not have a big influence on the algorithm, since it will just keep the evaluator busy in the first steps of the experiment. It actually does not, with a median number of evaluations of 11491 for initial population equal to 128 and 11675 for 256. Differences are not significant as indicated by Wilcoxon test. However, if we look at the running time (shown in figure **??**) there is a significant difference (at the 95% level, as indicated by the Wilcoxon test), with a median of 68 seconds ($i_p = 256$) vs. 63 ($i_p = 128$). Since this difference is not due to the number of evaluations, there must be other factor that creates it.

Several possible factors were examined: the actual size of the evaluator packet was essentially the same, indicating that there was no lack of chromosomes to evaluate. The conflicts in the reproducer had no difference either (there was a significant difference at the 15% level using t-test), and there was no difference between the sleeping periods spent by reproducers in both configurations. What was, then, the origin of the difference? To understand it we plotted the accumulated population generated by the reproducer, shown in figure **??**, it shows a small, but clear, difference. First we see that the solid line (representing smaller initial population, 128) runs for more steps, even as we know that, on average, it takes less, so each step takes shorter on average;

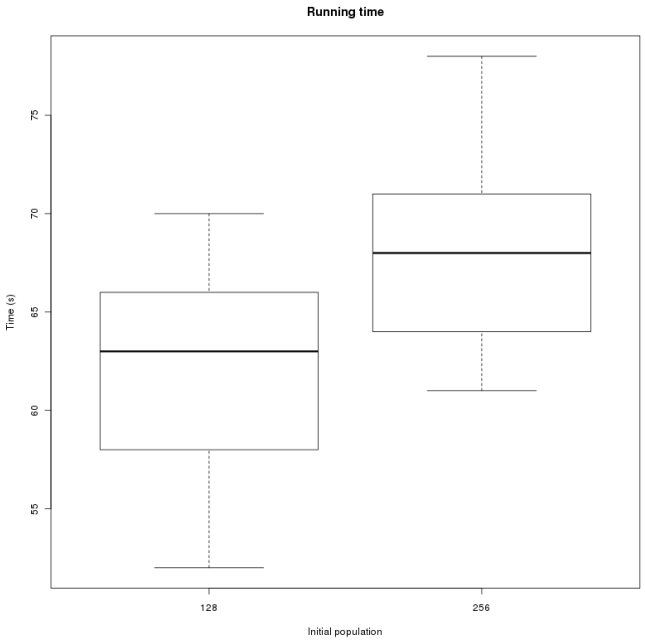

Figure 1: Boxplot of running time (in seconds) for different initial populations.

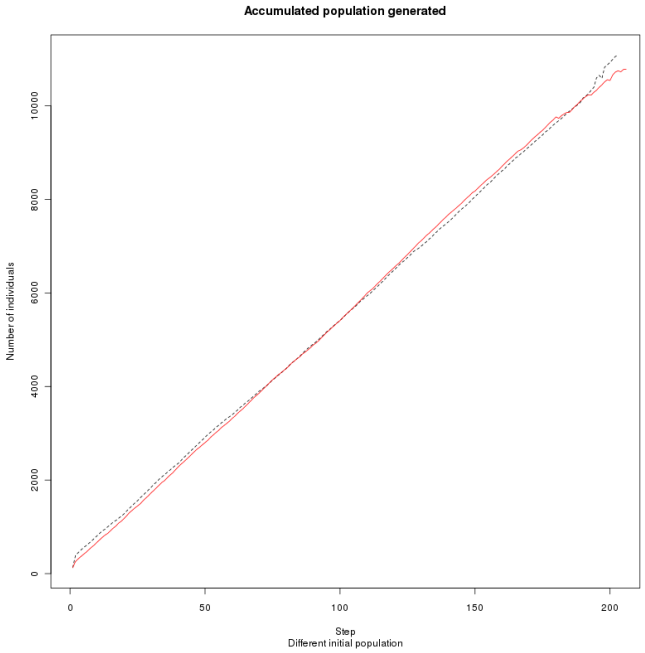

Figure 2: Accumulated population evaluated for initial population 256 (black, dashed) and 128 (red or clear, solid). x axis shows steps, although average time was smaller for the smaller population. This plot is the average of 10 experiments.

since we see that, at the beginning of the experiment, more population is generated when with initial population = 256, this probably implies that the smaller packet size used does have a small influence in running time; since there are hun-

dreds of steps in each experiment the accumulated effect is noticeable.

This experiment highlights two issues: first, that running time can be improved by tuning the chromosome supply, including the initial one; second, that an online evaluation of the algorithms must be made by looking at the evolution of variables such as accumulated population evaluated or generated, not only at the final values such as total number of chromosomes generated or evaluated.

However, there are other factors influencing the results; we will examine next the role of evaluation/reproduction block size in the results.

4.2 Block size

The division of the population in increasingly small block sizes should bring increases in speed, since there are more computing units in the evaluation/reproduction *pipeline*. Keeping the population (the sum of the block sizes of all clients) constant, we will increase first the number of evaluators and reproducers. Results are shown in figure **??** for running

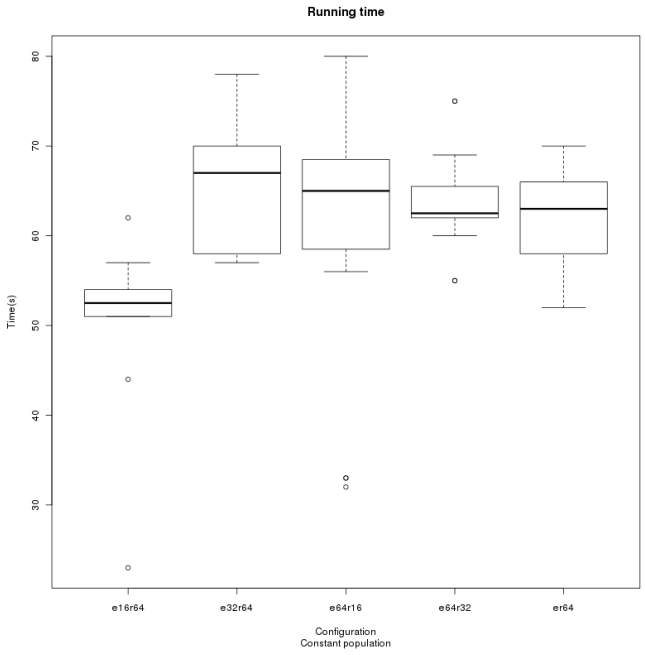

Figure 3: Boxplot of running time, in seconds, for the 5 evaluated configurations; e indicates the evaluator and r the reproducer block size. Number of clients is equal $64/r$ or e.

time and **??** for number of evaluations. The only significance difference running time-wise is among the $e16r64$ (4 evaluators with block size=16) and the rest. Although there seems to be also a small difference in the number of evaluations, once again it is only signification among $e16r64$ and the rest.

The first result we would like to point to is that while evaluation block size is an implementation parameter mainly (how many individuals we evaluate in one database request), reproducer block size is rather a *model* parameter, since different population size will have an influence in the resulting diversity and thus the number of evaluations needed to so-

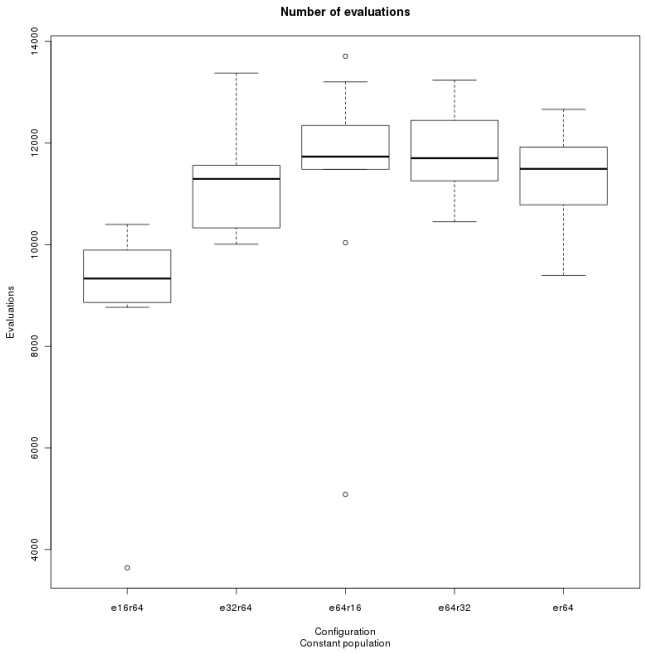

Figure 4: Boxplot of number of evaluations for the 5 evaluated configurations; e indicates the evaluator and r the reproducer block size. Number of clients is equal $64/r$ or e.

lution. Smaller populations will give less fit individuals less chances to reproduce, since it is more likely that the number of slots available to them will be rounded to 0, and thus increase selective pressure; bigger populations will be more diverse. This effect will be counter-balanced by the possible higher number of conflicts and lack of individuals to process due to the difference of speed among evaluators and reproducers. In fact, this is quite probably the mechanism that induces that result; in general, in research not reproduced here [**?**] we have proved reproduction is 50% faster, so, in general, there should be more individuals evaluated than reproduced. That is the reason why adding more reproducers just increases the imbalance among reproduction and evaluation. All these causes induce the observed effect: no change either in evaluations or running time, since reproducers keep chugging out new individuals that are not evaluated.

However, increasing the number of evaluators does have an effect, as shown in table **??**, which shows running times and average evaluations to solution for three of the above examined configurations.

Table 2: Varying block size experiment results. Values in boldface are both the best and significantly different.

Configuration	Running time	Evaluations
$E64R64$	62.10	11320
$E16R64$	**50.20**	**8910**
$E64R16$	62.50	11320

4.3 Database partitioning

To avoid having the same individual being evaluated twice, most approaches partition the space, having every client ac-

cessing one, and just one, partition. However, in order to do that, the number of clients must be known in advance, and besides no client could be dropped or a part of the space will be left without evaluation. So, as stated in the description of the algorithm (section **??**), one of the choices we made to partition the set of individuals in a decentralized fashion was to select the individuals to evaluate based on a random number. Every individual was assigned a random number when created. When the client requests a block, it generates a random number and it retrieves, at most, b individuals sorted in ascending order and starting with the one whose random key is the closest one generated. There are then two reasons why the individuals actually retrieved will be less than b: one, the number is very high and there are not enough individuals whose random key is bigger than the one generated, and second, there simply are not enough individuals left. This is usually not a problem for the evaluator, since the reproducer is faster, so the main reason will be the first. However, we cannot simply reduce the range for random numbers, since it would increase overlap among retrieved blocks.

Is this really a problem? Yes, increasingly so with block size, as shown in figure **??**. The fact that around 60% of the

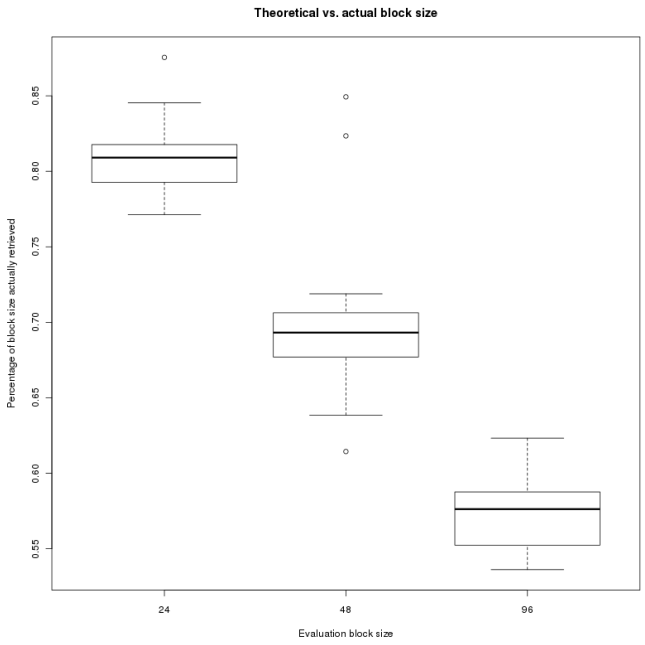

Figure 5: Percentage of block size actually retrieved depending on block size, for evaluator block size equal to 24, 48 and 96. All differences are significant.

$b = 96$ example are retrieved means that, on average, around 50 individuals are actually evaluated, and that before we take into account conflicts. It does not make a lot of sense to generate lots of requests to retrieve and evaluate just a few individuals, so we made a small adjustment to the random number range. Instead of using the whole random range from 0 to 1, we took into account the relationship among the block size b and the base population size p. Although the available number of individuals to evaluate will usually be larger, that will give us a rule of thumb of how many we should expect to be available. The random number was then generated uniformly in the range $[0, 1 - b/p]$. This might have the effect of changing block size for evaluation conflicts, but since the problem actually shows up when the block size is bigger, in which case the number of evaluators is small or even one, we might actually observe an improvement.

We tested this new strategy with the bigger block size, and the results can be observed in figure **??**, which shows the running time of two different configurations with full and reduced random range. There is very little difference

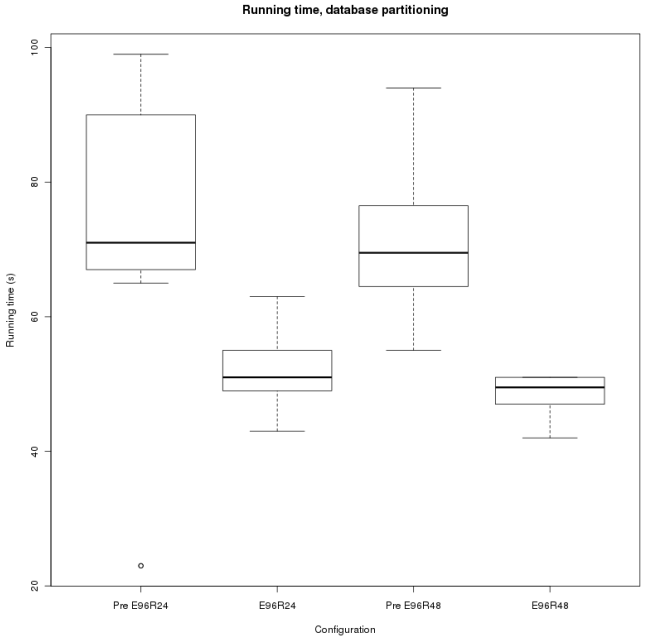

Figure 6: Boxplot of running time using the whole range for random number generation (labeled `Pre`) and with adjusted range. Other than that, configurations pre- and post are the same: evaluator block size 96 and reproducer block size equal to 24 (left) and 48 (right).

when we change split a reproducer with block size =48 into two different ones (differences, in fact, are not significant, it only seems to make result more predictable), but there is a significant difference when the random generation range is changed, from an average 73.73s to 52.55, almost 30% improvement (E96R24; the other result is not mentioned since it is not significantly different). The number of evaluations follow exactly the same pattern: average number of evaluations from 12180 to 9088, which means that the reduction in running time comes actually from the improvement in the number of evaluations needed to find the solution. Since changing the random range is the only alteration, that must be the reason, but what is the mechanism that creates this improvement?

Once again, we look at the other measures in figure **??**: number of conflicts in the reproducer. Many conflicts indicate that they are generating all over again the same individuals, probably due to the fact that the reproductive pool they are using is the same (due to overlap when retrieving it, or simply low turnout). This is what we see here: when

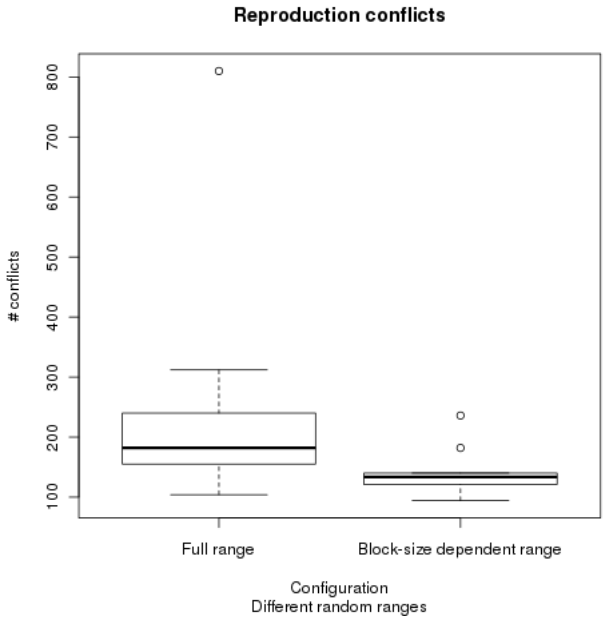

Figure 7: Boxplot of number of conflicts with full random range (left) and range dependent on block size (right). Difference is significant.

the *evaluator* random range is reduced, the average number of *reproductor's* conflict is reduced from 228.9 to 139.8, almost by 40%. A higher number of individuals evaluated in every request means that the reproductive pool (from where reproducers pick it) changes faster, leading to a lower number of conflicts, higher diversity and the lower number of evaluations (and thus running time) we observe. In fact, the change of block size retrieved is not so big (although significant): the median improved from 0.5656 to 0.5223, a mere 8%, but its cumulative effect was enough to improve diversity and markedly improve the number of evaluations needed and running time.

In fact, since increasing the evaluation block size seems to have beneficial effects on the algorithm (at least with a single evaluator), we could wonder what would happen if we used a *greedy* evaluator that would retrieve all available non-evaluated individuals and return them evaluated. We tested that, with surprising results, shown in table **??**. Both

Table 3: Fixed block size vs. *greedy* evaluator

Configuration	Running time	Evaluations
$E96R64$	49.5 ± 3	$\mathbf{8891 \pm 318}$
Greedy + $R64$	47 ± 4	10132 ± 786

experiments take pretty much the same time (difference not significant), but the number of evaluations is much better for the non-greedy strategy. From the purely computational point of view, this means that the greedy strategy is sequentially faster; but from the model point of view it results in a worse algorithm. Besides, a greedy strategy is not compatible with using several evaluators (any additional evaluator would have, most of the time, nothing to evaluate) and is less fault tolerant in that sense. The conclusion is here that a fixed evaluator block size is better, although size matters and if the number of evaluators is known in advance as big a size as possible (keeping it under the *base* population size is advisable).

5. CONCLUSIONS, DISCUSSION AND FUTURE WORK

In this paper we have examined design choices in SofEA, a pool-based distributed evolutionary implemented using CouchDB. The impact of parameters such as the initial population, the number of clients and the number of chromosomes processed in each request, and how these chromosomes are selected from the population are studied, measuring their effect on running time and number of evaluations, and, after explaining the results obtained looking at implementation measures such as the number of conflicts or the number of chromosomes effectively processed, current choices for the algorithm are shown and validated. The underlying result is also that SofEA shows certain robustness across parameter values, works asynchronously and can continue working even if one of several clients stop doing it, since their operation is independent of each other.

Some other results of this experiment is that adding evaluators brings better speed-ups than adding reproducers; one with a proper block size is enough, and new reproducers do not have an effect either on running or evaluation time. This might point to a design flaw that will have be examined in the future. Adding evaluators whose aggregated *population* is smaller than the base population size usually speeds up the experiment, provided block size it kept between certain limits (not too small, not too big). Other than that, we have proved that SofEA can offer the basis for an scalable (using CouchDB replication), asynchronous, distributed and fault-tolerant evolutionary algorithm system.

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; it will depend, anyways, on the particular problem), 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. We would like also to thank reviewers for their helpful comments.