Validating Design Choices in a Pool-based Distributed Evolutionary Algorithms Architecture

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

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

ABSTRACT

This paper introduces SofEA, a pool-based architecture built over CouchDB for distributing evolutionary algorithms (EAs) across computer network in an asynchronous and decentralized way. Clients perform different functions (evaluation, reproduction, selection) which leads to a complex behavior that will be examined in this paper, looking for the values that yield the best performance.

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]

General Terms

Algorithms

Keywords

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

1. SOFEA: A POOL BASED EVOLUTIONARY ALGORITHM

In this paper we examine design choices in SofEA, a pool-based evolutionary algorithm designed for volunteer computing. Implementation, and its resulting model, are geared towards creating a system in which spontaneous collaboration is possible, with clients contributing just a few cycles to an experiment. CouchDB is a document database system for creating web-based applications that eschew the need of additional middleware. Besides, pool-based systems in which the population resides in a pool, and clients pick elements from that pool, process them, and leave them back in the pool are the most adequate model for this kind of systems, and can be implemented using CouchDB. However, design choices are not trivial, and its study is what we approach in this paper.

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.
ACM 978-1-4503-1178-6/12/07.

CouchDB (`http://couchdb.apache.org`) is an key-value store that uses JSON (JavaScript Object Notation, a text serialization of arbitrary data structures). 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. Mapping an EA to this system has to take into account its peculiar features and go with its grain to achieve maximum performance. That is why we map chromosomes to objects, using the string as key and fitness, the string again and a random constant as document. *Revisions* are a straightforward way to represent the state of the chromosome, going from 1 (created) to 2 (evaluated) to 3 (*dead*); this will be used to select those in a particular state.

We will select chromosomes to process requesting chromosomes ordered by the random constant. Elimination of the worst chromosomes will be done by the *reaper* client: it will keep the best p chromosomes, updating the rest to revision 3. The *evaluator* and *reproducer* will be the other two clients; they will have to operate on a pre-generated initial population. All clients will check a document keyed by *solution* to know when the experiment is finished.

There are several issues with this model which boil down to oversupply (generating too many chromosomes), starvation (too few) and conflicts (several clients generating the same individuals). In this poster we will check how design choices affect them. Parameters, code and data are available from `https://launchpad.net/sofea`.

2. EXAMINING THE BEHAVIOR OF SOFEA

Experimental parameters are as shown in table 1 unless told otherwise. Population size was chosen after comparing

Table 1: Common experiment parameters.

Parameter	Value
Repetitions	10
Chromosome size	128
Initial population	128

it with 256; even if this could seem an independent factor, it does have a small but noticeable influence, and it is probably due to the fact that it is generating less useless chromosomes; however, this highlights the fact that we should look at dynamic variables, not only the final result.

We also used different *block sizes* for evaluators and reproducers, using powers of two and starting by 64 down to 16; not all combinations were tested, evaluator block size

64 was tested with reproducer 64, 32 and 16 and the same for reproducer 64; as the block size went down, the number of clients went up; for instance, there were 4 evaluators for E16R64. This was the only configuration that showed a marked difference in running time and number of evaluations with respect to the rest; it is compared with other results in table **??**.

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

Finally we will examine the influence on block size on the actual number of individuals processed, which needs not be the same as requested. We found that the number of individuals retrieved was a increasingly lower proportion with block size, with only 60% of the block size retrieved when this was equal to 96. That is why we adjusted the range taking into account the relationship among the block size and population size, generating a random number uniformly in the range $[0, 1 - b/p]$. 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; improvement is around 30% and is due to an improvement in the number of evaluations. But, besides, the number of reproducer conflicts is also decreased 228.9 to 139.8, almost by 40%.

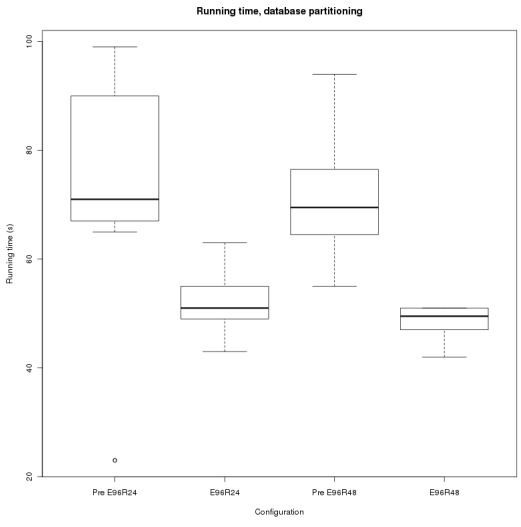

Figure 1: Boxplot of running time using the whole range for random number generation (labeled `Pre`) and with adjusted range. Evaluator block size = 96; reproducer block size = 24 (left), 48 (right)

In fact, since increasing the evaluation block size seems to have beneficial effects on the algorithm we tested *greedy* evaluator that would retrieve all available non-evaluated individuals and return them evaluated. Results are shown in table **??**. Both experiments take the same time (difference not significant), but the number of evaluations is much better for the non-greedy strategy. 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.

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

Configuration	Running time	Evaluations
$E96R64$	49.5 ± 3	**8891 ± 318**
Greedy + $R64$	47 ± 4	10132 ± 786

3. CONCLUSIONS AND DISCUSSION

In this paper we have examined design choices in SofEA, a pool-based distributed evolutionary implemented using CouchDB which was introduced in a previous paper [**?**]. 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.

Another result of this experiment is to check 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.

Acknowledgments

This work is supported by projects TIN2011-28627-C04-02 awarded by the Spanish Ministry of Science an Innovation and P08-TIC-03903 awarded by the Andalusian Regional Government.

4. REFERENCES

[1] Juan-J Merelo-Guervós, Antonio Mora, J. Albert Cruz, and Anna I. Esparcia. Pool-based distributed evolutionary algorithms using an object database. In Cecilia di Chio et al., editor, *EvoApplications 2012 Proceedings*, pages 441–450, 2012.