Using Free Cloud Storage Services For Distributed Evolutionary Algorithms

Maribel G. Arenas, Juan-Julián Merelo, Pedro Castillo
Juan-Luis J. Laredo, Gustavo Romero, Antonio M. Mora
University of Granada
Department of Computer Architecture and Technology, ETSIIT
18071 - Granada
maribel,pedro,gustavo@atc.ugr.es jmerelo,juanlu,amorag@geneura.ugr.es

ABSTRACT

Cloud computing, in general, is becoming part of the toolset that the scientist uses to perform compute-intensive tasks. In particular, cloud storage is an easy and convenient way of storing files that will be accessible over the Internet, but also a way of distributing those files and performing distributed computation using them. In this paper we describe how such a service commercialized by Dropbox is used for pool-based evolutionary algorithms. A prototype system is described and its peformance measured over a deceptive combinatorial optimization problem, finding that, for some type of problems and using commodity hardware, cloud storage systems can profitably be used as a platform for distributed evolutionary algorithms. Preliminary results show that Dropbox is indeed a viable alternative for execution of pool-based distributed evolutionary algorithms, showing a good scaling behavior with up to 4 computers.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; G.1.6 [**Mathematics of Computing**]: NUMERICAL ANALYSIS—*Optimization*

General Terms

Algorithms

Keywords

Cloud Computing, Cloud Storage, Evolutionary Algorithms, Distributed Algorithms

1. INTRODUCTION

The main problem in scientific computing nowadays is not to get computing power, but to aggregate it to solve problems. A typical department or lab (or, for that matter, a typical home) features half a dozen, or even more, CPUs that, if tapped, can be linked to perform computing experiments. In general, joining several different computers voluntarily in a metacomputer to perform an experiment is called volunteer computing [1, 2]. In this kind of computing, through the use of a downloaded client [3] or the browser [4] people (or the CPUs they own) participate in an experiment by letting these clients run (or by just visiting a web page).

Most volunteer computing environments work in a *star* configuration, by having a server with which all clients register, and letting this server send tasks to them periodically; clients then return results, which are logged and, if needed, checked. However, this implies that whoever is managing the experiment needs to run a server which, in occasions, can take a heavy load and be overwhelmed by it.

This server is mainly used for scheduling and balancing the tasks among the different clients; the network itself is used for communication, but all the interchange of information among clients must be cleared by the central server. However, one of the objectives of the work presented in this paper was to create an infrastructure that would get rid of this, by using the network itself to store and forward information.

The approach that we propose is to use *storage in the cloud* for providing information interchange [5]. Cloud storage outsources the repository of information by keeping it online, usually posing as a local resource (for instance, locally mounted filesystem) or via web services (whose use, anyways, can be wrapped to look like a local filesystem); a cloud storage service such as S3 [6], an offering from Amazon, can be used to interchange files between different clients without the person managing it having to operate a central server; in fact, there is such a central server, but it is a commodity which is paid by the user, instead of hosted by him or her.

This cloud storage service has other advantages: it is scalable, paid only when needed (and only as much as it is needed), and can be used by any device with a connection to the net. In fact, it can even be free, since there are several cloud storage services that, at an entry level, have no cost (and in that case usually called simply file hosting services, since that is its main function): Dropbox, Ubuntu One, Box.net, Sugarsync, Mozy and ZumoDrive, are some examples.

One of the most popular among them is Dropbox [7]. This service monitors designated local folders, and copies their content into central servers; then this content is replicated into the folders of whoever the folder owner shares the

GECCO'11, July 12–16, 2011, Dublin, Ireland.

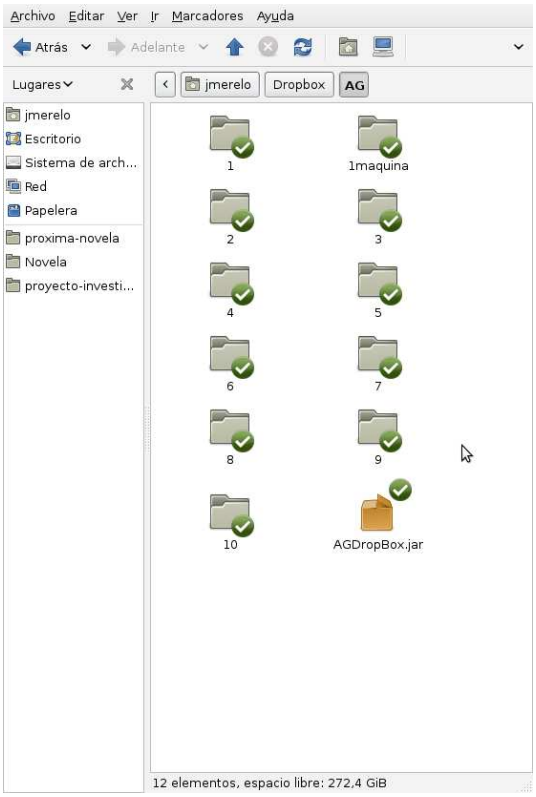

Figure 1: A folder monitorized by Dropbox in Nautilus, the Linux file navigator.

content with, or with other computers owned by the same person running the Dropbox client; an example of how the appearance of this folder is shown in Figure 1.

Dropbox can be seen as a *file/folder synchronization* service: folders shared among different clients, after a certain time has elapsed (which depends on the file size and bandwidth), will have the same contents. All the contents (files and folders) are either already synchronized or being synchronized, and the tick that is shown on the lower right corner of the icon reflects that fact by changing its appearance (either white tick on green, or rotating arrows on blue). The underlying architecture it uses is, besides plugins for integrating with local file navigators, a daemon or service that runs on the background and monitorizes when a file is being created or changed, and uploads it to the server; the same daemon receives requests from the Dropbox server and copies the information from the central server to the local folder. At any rate, Dropbox (like any other cloud storage or file hosting service) does have a central server, but from the client point of view the files are synchronized seamlessly from one client to the rest. This synchronization is not instantaneous (and is not effectively under the control of the user), but it takes at most 30 seconds and usually much less than that.

The fact that Dropbox runs on the background means that, from the client standpoint, writing to a shared folder is as fast as writing to the local filesystem (in fact, it is nothing more than that). File replication (or updating) is the slow operation, but that is seamless for the client. Another advantage is that, for a certain level of operation, it is free: up to 2 GB of storage (with some size bonuses obtained via inviting other users); other levels are also inexpensive. There is no upper bound on bandwidth, if used reasonably, on the server side; however, the clients can set it so as not to hog the home/office Internet connections.

That is why we set to use Dropbox as commodity file hosting service to perform distributed evolutionary computing experiments. Following the island-based model [8], the basic idea is to use a dropbox-monitored folder to drop the immigrants that will move to other islands, that is, other computers monitorizing this folder. Each computer will run independently, and will use that folder as a pool to drop and pick up individuals coming from other nodes. The main idea of this paper, which is basically a proof of concept, is to see what are the possibilities of this setup as a multicomputer by implementing an evolutionary algorithm using it, and then, measuring the speedup when several computers are used at the same time. In the spirit of the research, these will be off the shelf personal desktop or laptop computers, with different CPU, storage and possibly network capabilities and running different operating systems or versions of it. The problem we will attempt to solve is a hard combinatorial problem, so that it takes time enough to get some improvement from parallelization. For the time being, we will only try to measure how running time scales when new (heterogeneous) nodes are added to the system, being the main objective to test if this kind of file hosting systems are suitable for using them for scientific distributed computing.

The rest of the paper is organized as follows: next we will introduce the state of the art related to the research presented here, to be followed by a description of the algorithm, the experimental setup and the implementation in Section 3. The results will be presented in Section 4, to be followed by the discussion and future lines of work in Section 5.

2. STATE OF THE ART

Cloud computing [9, 10] is an emergent technology, and as such research related to it is just recently emerging. Research addressing cloud storage is mainly related to content delivery [11] or designing data redundancy schemes to ensure information integrity [12]. However, its use in distributed computing has not been addressed in such depth. Even if it is related to data grids [13], in this paper we address the use of free cloud storage as a medium for doing distributed evolutionary computation, in a more or less parasitic way [14], since we use the infrastructure laid by the provider as part of an immigration scheme in an island-based evolutionary algorithm [8].

Thus we will have to look at pool-based distributed evolutionary algorithms for the closest methods to the one presented 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 evolutionary algorithms. Asynchronous Teams or A-Teams [15, 16, 17] 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 working on them. This technique is not constrained to evolutionary algorithms, since it can be applied to any population based technique, 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 [18] system, which creates a data flow framework, with its own language (called ZigZag), which can be applied, in particular, to evolutionary algorithms.

While algorithm design is extremely important, implementation issues always matter, and some recent papers have concentrated on dealing with pool architectures in a single environment: G. Roy et al. [19] propose a shared memory multi-threaded architecture, in which several threads work independently on a single shared memory, 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 [20], who present a design pattern for persistent and distributed evolutionary algorithms; although their emphasis is on persistence, and not performance, 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 [4] 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. However, given the rise of cloud computing platforms over the last few years, interest in this kind of algorithms has bounced back, with implementations using the public FluidDB platform [21] having been recently published.

3. DESCRIPTION OF THE ALGORITHM AND IMPLEMENTATION

We will divide this section in several parts: firstly, it is commented the description of the algorithm itself; then the problems to be solved as tests are presented. Finally, we describe the decisions needed to implement it efficiently over the cloud storage service in the last subsection.

3.1 A distributed pool-based evolutionary algorithm over Dropbox

A pool based evolutionary algorithm can be described as an island model [22] without topology; in fact, it is closer to the *island* metaphor since migrants are sent to the *sea* (pool), and come also from it, that is, the evolutionary algorithm is a classic one with binary codification, except for two steps within the cycle that (conditionally) emit or receive immigrants. A maximum number of evaluations for the whole algorithm is set from the beginning; we will see later on how to control when this maximum number of evaluations is reached.

During the evolutionary loop, new individuals are selected using 3-tournament selection and generated using bit-flip mutation and uniform crossover Operator rates are set initially to 0.9 and 0.1, but if 500 generations without a change are reached, crossover is decreased and mutation increased by 0.1; both values are bound by 0.5 (minimum for crossover and maximum for mutation). This change to the baseline algorithm has to be applied in the hard (deceptive) combinatorial problems that have been used in this paper.

Migration is introduced in the algorithm as follows: after the population has been evaluated, migration might take place if the number of generations selected to do it is reached. The best individual is sent to the pool, and the best individual in the pool (chosen among those emitted by the other nodes) is incorporated into the population; if there has been no change in the best individual since the last migration, a random individual is added to the pool. Migrants, if any, are incorporated into the population along with the offspring of the previous generation using generational replacement with a 1-elite. Population was set to 1000 individuals for all problems, and the minimum number of evaluations has been four million, which is enough to find the solution to the problems. Migration was performed after every 250 generations.

The partial results are updated at the end of the loop to check if the algorithm has finished. Since all the nodes act asynchronously due to their different capabilities, the number of local evaluations might vary; the condition for finishing is reaching a certain number of global evaluations. In fact, due to the conditions of the method, this number is a minimum, since the fact that it has been reached is not propagated instantaneously to all nodes.

One of the advantages of this topology-less arrangement is the independence from the number of computers participating in the experiment, and also the lack of need from a *central* server, although it can be arranged so that one of the computers starts first, and the others start running when some file is present. Adding a new computer, then, does not imply to arrange new connections to present computers; the only thing it needs to do is to locate the directory that is shared through Dropbox.

3.2 Fitness functions: Trap and MMDP

Two representative functions have been chosen to perform the tests. The main idea is to chose them to be difficult, and thus taxing for a distributed evolutionary algorithm. Besides, these functions are usually considered as a benchmark for evaluating the quality of algorithms, making then easy to compare our results with others (or even our own).

A trap function [23] is a piecewise-linear function defined on unitation (the number of ones in a binary string). There are two distinct regions in the search space, one leading to a global optimum and the other leading to the local optimum (see Figure 2). In general, a trap function is defined by the following equation:

$$trap(u(\overrightarrow{x})) = \begin{cases} \frac{a}{z}(z - u(\overrightarrow{x})), & \text{if } u(\overrightarrow{x}) \leq z \\ \frac{b}{l-z}(u(\overrightarrow{x}) - z), & \text{otherwise} \end{cases} \quad (1)$$

where $u(\overrightarrow{x})$ is the unitation function, a is the local optimum,

b is the global optimum, l is the problem size and z is a slope-change location separating the attraction basin of the two optima.

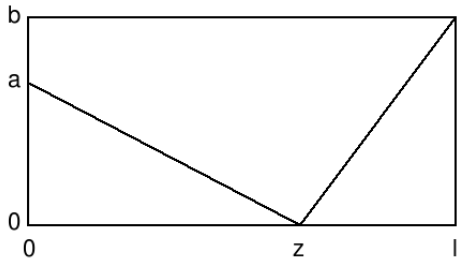

Figure 2: Generalized *l-trap* function.

For the experiments, 4-trap functions were designed with the following parameter values: $a = l-1$, $b = l$, and $z = l-1$. Unlike 2 and 3-trap, 4-trap is fully deceptive. This function has been used in other papers [24] to measure scalability; in this case it has been chosen mainly for its difficulty and the fact that, for the number of evaluations we have set, it is not usually able to find the solution (which would skew scalability results). The number of *traps* we have used in these experiments is 30, once again, rather a high number to avoid convergence.

On the other hand, the MMDP [25] is a deceptive problem composed of k subproblems of 6 bits each one (s_i). Depending of the number of ones (unitation) s_i takes the values depicted next:

$$fitness_{s_i}(0) = 1.0 \qquad fitness_{s_i}(1) = 0.0$$
$$fitness_{s_i}(2) = 0.360384 \qquad fitness_{s_i}(3) = 0.640576$$
$$fitness_{s_i}(4) = 0.360384 \qquad fitness_{s_i}(5) = 0.0$$
$$fitness_{s_i}(6) = 1.0$$

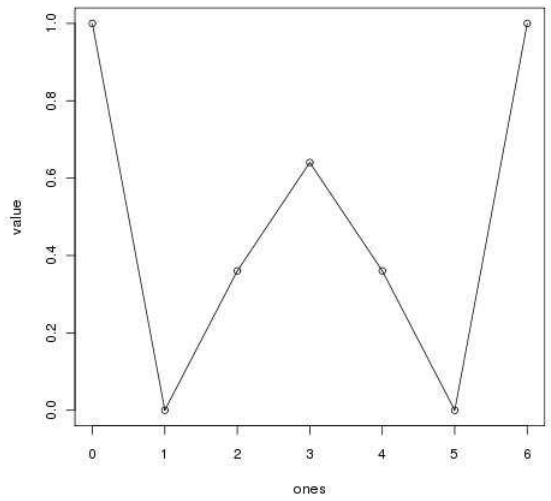

Figure 3: Representation of one of the variables of the MMDP problem.

The fitness value is defined as the sum of the s_i subproblems with an optimum of k (equation 2). Figure 3 represents one of the variables of the function. The number of local optima is quite large (22^k), while there are only 2^k global solutions.

$$f_{MMDP}(\vec{s}) = \sum_{i=1}^{k} fitness_{s_i} \tag{2}$$

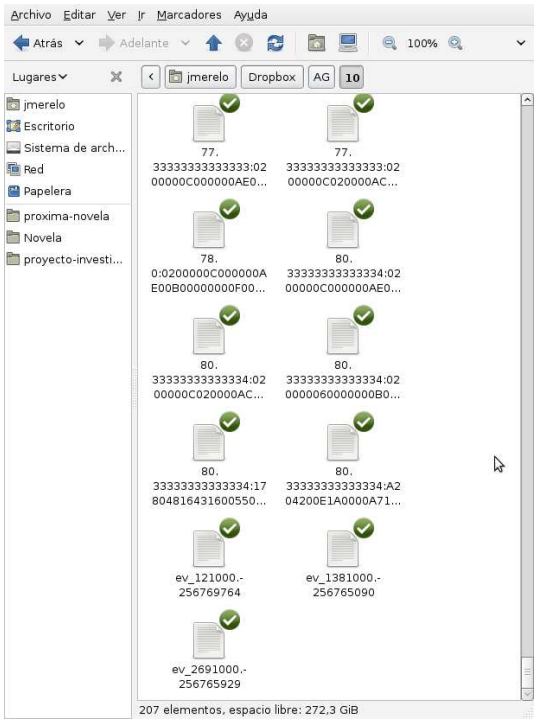

Figure 4: Immigrants in a Dropbox folder, with fitness, chromosome and original node codified as file names. At the bottom, the three files which are used to show the number of evaluations per node are pictured.

3.3 Cloud storage implementation

The experimental result will be aimed at reducing the overhead of using the filesystem for communication among nodes, and taking into account that all such communication must be done using that kind of files. There will be a single folder per experiment, which is kept or removed for performing other experiments on it. The folder has to be shared among all the participants in the experiment, either by using a single user or by applying the *share* mechanism in Dropbox folders, which allows sharing of folders among different users. In fact, this was what we used in these experiments; there was a single `AG` folder (shown in Figure 1) shared among the participants, and different subdirectories were used to perform each experiment.

One of the main implementation decisions was to use filenames, instead of file contents, to codify the individuals that are migrated. Each individual is codified in a file with size 0 whose name includes the fitness, the binary chromosome in hexadecimal and the random seed used to start the algorithm (and which is used as an unique ID); this is shown in figure 4. This means, in fact, that what is being read and written among the different nodes is the directory file, not the files themselves, making the algorithm faster, sav-

ing bandwith and obviously, avoiding storage limit within the free option we are using. The file itself does not have to be read, saving also time and reducing overhead. This implies a limitation on the chromosome size and also the precision of the fitness, but for binary problems like this one does not really mean much. Besides, it is very easy to overcome that limitation by just including information within the file using some suitable codification. The codification has an added advantage: it is unique per node, which avoids re-immigrating those individuals that have been sent to the pool, increasing diversity for the low price of including an unique ID per node.

A mechanism was created to start execution more or less synchronously. There is one computer that starts the experiment, and the others follow by monitoring the existence of a file in the folder; when that file is created, the execution starts. This means a small delay (the delay in synchronizing both computers and the small delay from the detection of the existence and the start of the new algorithm). Although that mechanism might be needed for many computers, or just unattended operation, in these experiments we started all nodes more or less at the same time by pressing the start key simultaneously.

Since all nodes are heterogeneous, each one runs the evolutionary algorithm on its own schedule. After every generation, a file is written including in the filename the random seed and the number of evaluations performed. It also checks for all files of the same kind in the folder or directory, checks for the highest number of evaluations in all of them, adds them up and finishes if the minimum number of evaluations has been reached; if it has, it stops. These files are shown, among the rest of the files representing immigrants, in figure 4; the three files show that the three nodes have performed 0.122, 1.38 and 2.69 millions of evaluations each.

Please note that except for the beginning and the end, which happen more or less synchronously, the algorithms are carried out asynchronously; depending on capacity, one node might carry out twice or three times as many generations as the other (or others). This leads to interesting algorithmic effects [26], but they have not been studied in this paper, that concentrates in running time, and will be addressed in future works.

4. EXPERIMENTS AND RESULTS

First we will describe how the evolutionary algorithm was adapted to using Dropbox as a pool, and then we will show the results obtained with it.

4.1 Experimental setup

The experiments were performed with 4 machines of the following characteristics:

- Two of the computers were Sony VAIO VGN-SR29VN with a Intel Core2Duo at 2.4 GHz, running Ubuntu 10.04 and Java version 1.6.0_20. These were used as the first and third computer in the set.
- The second computer added to the set was VAIO with an Intel Core i7 with Ubuntu 10.04.
- The fourth computer was also a VAIO, running Windows 7 and with an Intel core i5.

The computers were connected to the Internet using WiFi, the `cviugr-v2` Campus-wide wireless connection, which uses WPA/Enterprise encryption. This probably could have some influence in the bandwith used, but in particular in this case we think that we used it well under capacity. In any case, the idea was to use all the computers with the same type of connection, to avoid to introduce another variable in the final analysis of results.

The version of Dropbox running was updated to the latest by Feb 5th, 2011 (0.7.110, Linux version). We used the default configuration for all of them, which does not have any limitation on upload or download rate, chooses automatically the best upload rate and does not use the proxy. The network was a mixture of wired and wireless network in the University of Granada; both are different sub-networks, so there is at least one switch among them.

The time shown in the results is wallclock time (as opposed to CPU time, the actual time the program has been running in the CPU), as measured by the difference among the end and starting time in the timestamps attached to the files.

4.2 Results

The experiments were performed first on a single computer, then on two, three and finally, four computers, organized as said above. The first thing considered was that, effectively, the evolutionary algorithm was working by checking the evolution of the fitness, as reflected by the immigrants in the pool; remember that the best individual in the current generation is deposited in the Dropbox folder. Taking into account that the file has a timestamp, no other logging is needed; the filesystem metadata reflects the time at which that file has come into existence, and thus the state of the evolution at that point in time. Times were taken over the "original" computer, that is, the first that initiated the experiment; although there is nothing special with that machine. The timestamps for the files originated in that computer are original, the others will have some delay over its creation, and can also be synchronized out of order. To smooth over these differences, we did a bandplot, with the resulting line shown in figure 5. This figure, which plots fitness versus time for 10 runs, reflects the fact that when there is no improvement over the previous generation, a random immigrant is sent to the pool. At the beginning, the average fitness is rather high since only the bests are sent to the pool, after that, random individuals are sent to it, and thus, the average fitness goes down. Maximum fitness does have a different behavior, increasing (on average) until the 30th second, and then staying more or less at the same level. From this graph several conclusions can be drawn: first, that most executions end after 60 seconds (more sparse dots after that point); second, most improvement happens at the beginning of the run, with improvements happening much more slowly after the first third, and finally the population in the two machines becomes very much alike after a few generations, since the bands come close together.

The most important result, however, is the scaling that can be obtained when several computers are trying to solve the same problem and interchanging solutions through the pool. In figures 6 and 7 we plot the average duration of a single evaluation, computed by dividing the time taken for every run by the total number of evaluations performed in the run, and then averaging over all runs. The plot shows that the average time needed for a single evaluation decreases almost fourfold for Trap, that is, close to a linear speedup,

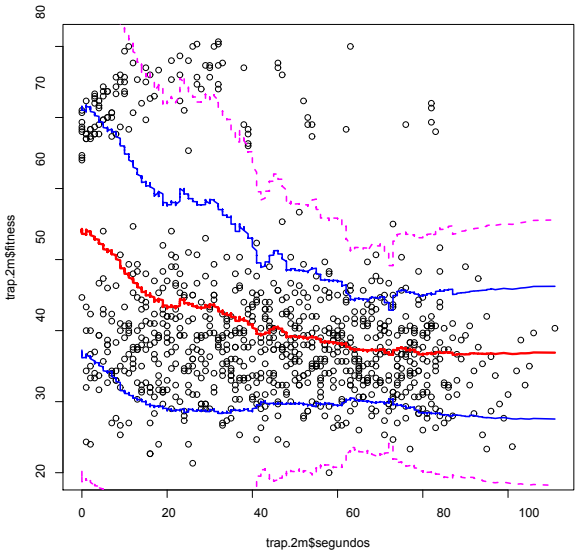

Figure 5: Bandplot Fitness vs. Time (in seconds) graph; x axis reflects the time in seconds since the start of the simulation, y axis the fitness for the Trap function and the 2-machines experiment. Bands show the locally smoothed mean and standard deviation (using the `bandplot` function from the `gplots` R package).

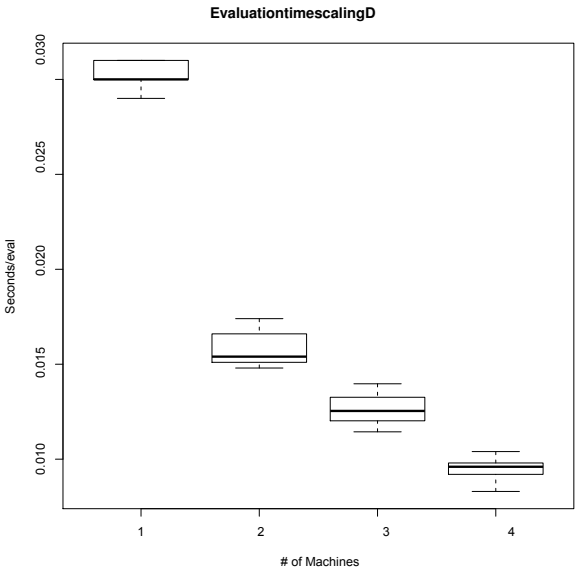

Figure 6: Scaling of the time needed to perform a single evaluation, as a boxplot against the number of machines, for the MMDP function.

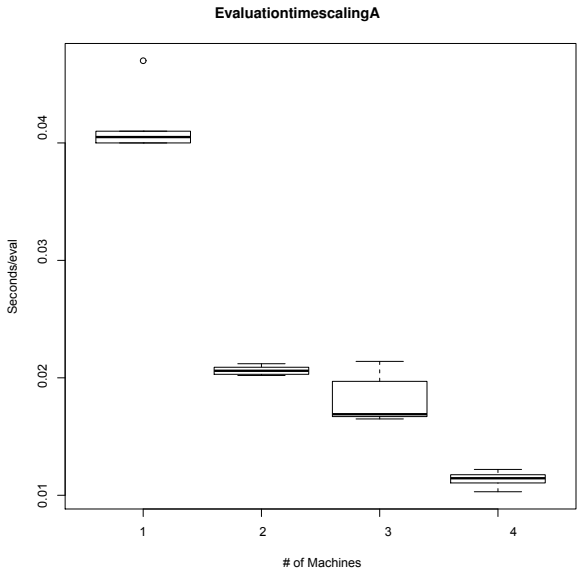

Figure 7: Scaling of the time needed to perform a single evaluation, as a boxplot against the number of machines. Top: Scaling for MMDP; bottom for the Trap Function.

while the decrease is of around 60% for the MMDP function. These results are quite good, considering that the computers are different, and the network is not exclusive. It would be taken into account that the total number of evaluations in each case is bigger than the maximum, and becomes bigger when the number of machines is increased; while with a single computer the number of evaluations is exactly equal to the maximum, with four computers it goes up to around 25% more, since the fact that the minimum number of evaluations has been reached takes some time to reach them, during which, of course, they have performed more evaluations.

The scaling graph also points to the fact that load balancing is done automatically by the simple device of letting every computer run until the files that show the number of evaluations done add up to the minimum. That does not mean that every new node joining the evolution at any moment really has an influence on the final result, as shown in [26], but at least from the point of view of adding raw evaluations to the experiment, they are giving as many as they can, without any gaps due to synchronization. In fact, this is common to all asynchronous distributed evolutionary algorithms [27].

5. CONCLUSIONS AND FUTURE WORK

By performing some experiments using laptop computers and the free service level of the Dropbox file-storage and sharing system, in this paper we have proved that these kind of cloud storage systems can be used profitably for distributed evolutionary algorithms, without the need to acquire or set up complicated cloud or grid infrastructure. Using tools available in every classroom, lab, office or home and freely available over the internet, a multicomputer running an evolutionary algorithm that has a good scaling behavior

can be set up to perform heavy-duty evolutionary computation experiments.

To demonstrate it, an island-based pool evolutionary algorithm has been implemented over the Dropbox system, and two well-know test problems (such as some Trap Functions and MMDP) have been addressed. The gain obtained in the two instances of the problem considered lies between 3 and 4 for 4 computers, which is indeed promising.

The code and datasets used to perform these experiments will be made available as open source software, so that results can be easily reproduced and/or extended to perform other types of experiments, or to adapt to other file sharing services (free or premium).

On the other hand, a lot of work remains to be done. The first is to study the algorithmic effect of this kind of pool architecture, and to check the best policies for migration to and from this pool; migration rates and other evolutionary algorithm parameters will have to be tuned so as to obtain the best number of average evaluations to solution. An extensive comparison with other types of distributed evolutionary frameworks is in order, either from the algorithmic or the implementation point of view.

Finally, the upper bounds to this scaling will have to be measured too. These will probably arise from the free nature of the file sharing service used, but this one as well as more structural ones (like the limitations of using directories as store for individuals and the locking effects due to that) will have to be measured.

Acknowledgements

This work has been supported in part by the CEI BioTIC GENIL (CEB09-0010) MICINN CEI Program (PYR-2010-13) project and the Andalusian Regional Government TIC-03903 and P08-TIC-03928 projects.

6. REFERENCES

[1] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 196–203, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Daniel Lombraña Gonzalez, Francisco Fernández de Vega, Leonardo Trujillo, Gustavo Olague, F. Chavez de la O, M. Cardenas, Lourdes Araujo, Pedro A. Castillo, and Ken Sharman. Increasing gp computing power via volunteer computing. *CoRR*, abs/0801.1210, 2008.

[3] D.L. Gonzalez, F.F. de Vega, L. Trujillo, G. Olague, L. Araujo, P. Castillo, J.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*, pages 419–423, Feb. 2009.

[4] J.J. Merelo, P.A. Castillo, J.L.J. Laredo, A. Mora, and A. Prieto. Asynchronous distributed genetic algorithms with Javascript and JSON. In *WCCI 2008 Proceedings*, pages 1372–1379. IEEE Press, 2008.

[5] Wikipedia. Cloud storage — Wikipedia, The Free Encyclopedia, 2011. [Online; accessed 5-February-2011].

[6] J. Varia. Cloud architectures. White Paper of Amazon, 2008.

[7] Wikipedia. Dropbox (service) — wikipedia, the free encyclopedia, 2011. [Online; accessed 5-February-2011].

[8] E. Cantú-Paz. Topologies, migration rates, and multi-population parallel genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO-99*, pages 13–17, 1999.

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.

[10] Hai Jin, Shadi Ibrahim, Tim Bell, Wei Gao, Dachuan Huang, and Song Wu. Cloud types and services. In Borko Furht and Armando Escalante, editors, *Handbook of Cloud Computing*, pages 335–355. Springer US, 2010.

[11] James Broberg, Rajkumar Buyya, and Zahir Tari. Metacdn: Harnessing [']storage clouds' for high performance content delivery. *Journal of Network and Computer Applications*, 32(5):1012 – 1022, 2009. Next Generation Content Networks.

[12] Lluis Pamies-Juarez, Pedro García-López, Marc Sánchez-Artigas, and Blas Herrera. Towards the design of optimal data redundancy schemes for heterogeneous cloud storage infrastructures. *Computer Networks*, In Press, Corrected Proof:–, 2010.

[13] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187–200, 2000.

[14] Albert-Lászlo Barabási, Vincent W. Freeh, Hawoong Jeong, and Jay B. Brockman. Parasitic computing. *Nature*, 412(6850):894–897, August 2001.

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

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

[17] S. Talukdar, S. Murthy, and R. Akkiraju. Asynchronous teams. *INTERNATIONAL SERIES IN OPERATIONS RESEARCH AND MANAGEMENT SCIENCE*, pages 537–556, 2003.

[18] X. Llorà, B. Ács, L.S. Auvil, B. Capitanu, M.E. Welge, and D.E. Goldberg. Meandre: Semantic-driven data-intensive flows in the clouds. Technical Report 2008103, Illinois Genetic Algorithms Laboratory, 2008.

[19] G. Roy, Hyunyoung Lee, J.L. Welch, Yuan Zhao, V. Pandey, and D. Thurston. A distributed pool architecture for genetic algorithms. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 1177–1184, May 2009.

[20] A. Bollini and M. Piastra. Distributed and persistent evolutionary algorithms: a design pattern. In *Genetic Programming, Proceedings EuroGP´99*, number 1598

in Lecture notes in computer science, pages 173–183. Springer, 1999.

[21] J.J. Merelo. Fluid evolutionary algorithms. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[22] D. Whitley, S. Rana, and R. Heckendorn. Island model genetic algorithms and linearly separable problems. *Evolutionary Computing*, pages 109–125, 1997.

[23] David H. Ackley. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[24] Juan Luis Jiménez Laredo, A. E. Eiben, Maarten van Steen, and Juan Julián Merelo Guervós. EvAg: a scalable peer-to-peer evolutionary algorithm. *Genetic Programming and Evolvable Machines*, 11(2):227–246, 2010.

[25] David E. Goldberg, Kalyanmoy Deb, and Jeffrey Horn. Massive multimodality, deception, and genetic algorithms. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, 2*, pages 37–48, Amsterdam, 1992. Elsevier Science Publishers, B. V.

[26] Juan J. Merelo, Antonio M. Mora, Pedro A. Castillo, Juan L. J. Laredo, Lourdes Araujo, Ken C. Sharman, Anna I. Esparcia-Alcázar, Eva Alfaro-Cid, and Carlos Cotta. Testing the intermediate disturbance hypothesis: Effect of asynchronous population incorporation on multi-deme evolutionary algorithms. In Gunter Rudolph, Thomas Jansen, Simon Lucas, Carlo Poloni, and Nicola Beume, editors, *Parallel Problem Solving from Nature - PPSN X*, volume 5199 of *LNCS*, pages 266–275, Dortmund, 13-17 September 2008. Springer.

[27] B. Baran, E. Kaszkurewicz, and A. Bhaya. Parallel asynchronous team algorithms: Convergence and performance analysis. *IEEE transactions on parallel and distributed systems*, 7(7):677–688, 1996.