# Assessing Different Architectures for Evolutionary Algorithms in JavaScript

Juan-Julián Merelo,
Pedro Castillo,
Antonio Mora
GeNeura, ETSIIT + CITIC, U. Granada
jmerelo,pedro,amorag@geneura.ugr.es

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

Víctor M. Rivas-Santos
Universidad de Jaén
vrivas@ujaen.es

## ABSTRACT

JavaScript (JS) is nowadays the only language that can be used to develop web-based client-server applications in both tiers, client and server. This makes it an interesting choice for developing distributed evolutionary computation experiments, but the best way from algorithmic and practical point of views is not clear, so we will compare different distributed EC architectures in JavaScript using NodEO, an open source JS framework released by us.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

## Keywords

JavaScript; node.js; parallel evolutionary algorithms; asynchronous evolution

## 1. INTRODUCTION

`node.js`, which is sometimes calles simply Node, is a JS interpreter whose default input/output mode is asynchronous. In the context of a distributed evolutionary algorithm using migration, this means that when performing network operations like migration, we do not know when it will eventually finish or even if it will be performed in the exact sequence we called them. This implies a change in the evolutionary algorithm whose impact in performance will have to be evaluated. This will happen across different distributed evolutionary algorithm architectures, but they will be impacted in a different way. In this paper we will check the increasingly popular peer to peer distributed EA architecture along the more classical client-server architecture. We will do it using the NodEO open source evolutionary algorithm library written in Node [2], which is available in any node instance via the Node Package Manager at `https://npmjs.org/package/nodeo` with a GPL license and whose development is open at GitHub. This library will use the `express.js` server and `restler` client modules to implement network operations using REST commands.

The function chosen for doing all the experiments is a classical deceptive function called Trap [1]:

$$T(x) = \begin{cases} a*(z-x)/z & \text{if } x <= z \\ b*(x-z)/(l-z) & \text{if } x > z \end{cases}$$

where $x$ is the number of ones in the block, $l$ is the length of the block, and $a$ and $b$ are two constants that meet the condition $a < b$. In this experiment we will use $z = 3, a = 1, b = 2$.

After some initial experiments to find the population size we will test two different architectures: *P2P* in which every *node* (which we will call process from now on, since they are implemented as such and to avoid confusion with `node.js`, the JS interpreter) communicates with the rest and needs to know the directions of the other nodes to interchange information, which is done through HTTP `PUT` requests; and *pool-based* where communication of one client with other is only done through the server, with each program acting independently and knowing only about this server. This makes easy to add new clients, but also turns the single server into a possible bottleneck.

Some initial tests changing the generation gap (or migration rate) shows that waiting for 20 generations before migration is a reasonable quantity, and it will the one used in the experiments. Besides, we have tested up to 16 processes in a single machine with no noticeable degradation in performance.

However, this programming effort can be used in a different direction, and that is what we have attempted with the pool architecture, with clients all working against a single server. In principle we wanted to test the same architecture with the same generational gap, that is, total population for all nodes equal to 512, with this population divided among processes. However, since all the requests are done to a single process its event queue saturates very fast which led us to increase the generational gap with the number of processes; even so, it brings errors which crash the clients in some cases.

In general and from the point of view of the operating system load, no great change is observed with the addition of one process ($n$ client processes + server) to the pool; if there is any difference in time, it should not be attributed to increased OS load or, for that matter, to the small changes

*GECCO'14,* July 12–16, 2014, Vancouver, BC, Canada.
ACM 978-1-4503-2881-4/14/07.
http://dx.doi.org/10.1145/2598394.2598460.

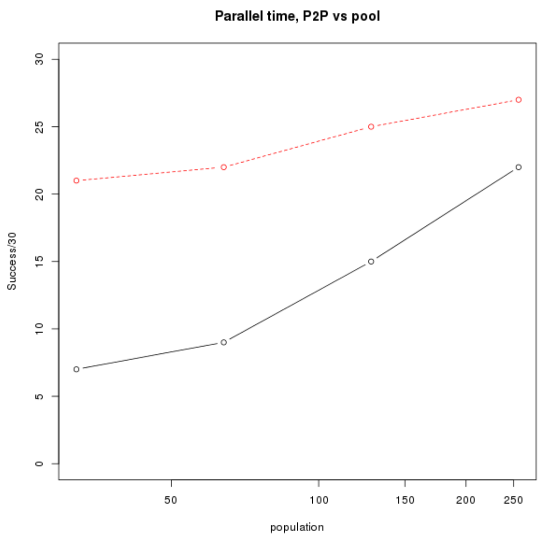

**Figure 1: Average time, in milliseconds, for successful runs. Black, solid represents the previously mentioned P2P architecture, while the light, dashed line represents the pool-based, single-server architecture.**

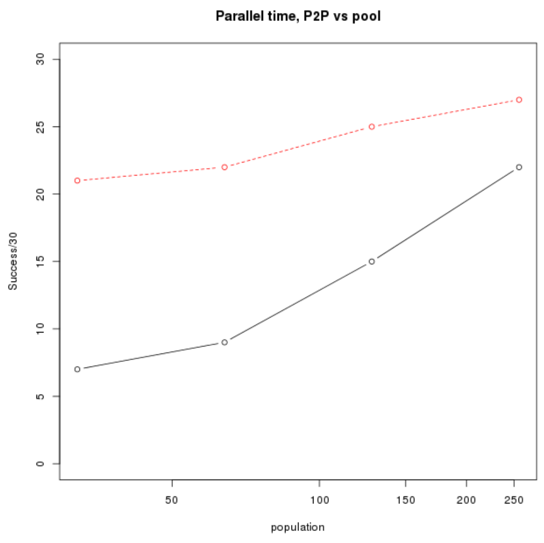

**Figure 2: Number of successful runs out of 30. Black, solid line corresponds to the P2P architecture, light-colored, dashed corresponds to the single-server architecture (pool).**

in the application architecture done. Even if the P2P applications do a single request (a GET) and this one performs two (first a PUT and then a GET), it probably cancels out with the fact that the P2P processes must also *respond* to requests from time to time. In fact, what we observe in the comparison of both types of architectures in Figure 1 is that there is a difference for the smallest number of processes and the biggest number of processes but they go in different directions, so it is difficult to say if, in general, there is any difference (there is none for $p = 64, 128$) and what is its origin.

But a more dramatic change of scenario is shown in Figure 2, which shows that the success rate is noticeably higher for the pool-based architecture, although decreasingly so with the increasing population; this is only to be expected since, in fact, success rate increased with the population in the P2P architecture. This makes this pool-based architecture from the algorithmic point of view, the best alternative. Besides, they are not mutually exclusive.

This conclusion is reached on top of having proved (yet again, some might say) that JavaScript, its implementation in `node.js`, and the simple library called NodEO we are presenting in this paper, are valid platforms for performing distributed computation experiments, since they allow to create rapid prototypes to concentrate on system architecture and the solution of problems via evolutionary algorithms. In an unconstrained environment, JavaScript will probably be slower than Java or C++, although its speed is on par with other scripting languages like Ruby. However, in an environment such as a multi-tier architecture that includes rich Internet applications (with an UI written in JavaScript in the browser) or even mobile applications (which can easily be done in JavaScript via the PhoneGap framework or simply HTML5 in any browser) JavaScript can offer an excellent performance and even algorithmic advantages in distributed evolutionary algorithms, as proved here.

## 2. CONCLUSIONS

In this paper we have proved the validity and performance `node.js`-based distributed evolutionary algorithm by using a standard module, NodEO. `node.js` is better known in development and open source circles than in the scientific community, so our intention was to introduce it to the evolutionary computation (EC) community by proving its value as a platform for EC experiments. A basic EC library has been created and released, so it is available to the researches. The library can be expanded and, being open source, can be adapted and suited to the needs of any particular user; due to the expansion of the JavaScript and `node.js` community, it should be increasingly easy to find people interested and skilled enough to work in evolutionary algorithms using JavaScript and `node.js`, and, from the other end, it could get the `node.js` community interested in our area, which might prove the source of interesting problems that can be dealt with from the point of view of metaheuristics.

## 3. ACKNOWLEDGMENTS

Funded by CEI-BioTIC grant CANUBE (CEI2013-P-14) and ANYSELF project (TIN2011-28627-C04-02).

## 4. REFERENCES

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

[2] J. J. Merelo Guervós. NodEO, a evolutionary algorithm library in Node. Technical report, GeNeura group, Mar. 2014. Available at `http://figshare.com/articles/nodeo/972892`.