Performance improvements of evolutionary algorithms in Perl 6

Juan-Julián Merelo-Guervós
Universidad de Granada
Granada, Spain
jmerelo@ugr.es

José-Mario García-Valdez
Instituto Tecnológico de Tijuana
Tijuana, Mexico
mario@tectijuana.edu.mx

ABSTRACT

Perl 6 is a recently released language that belongs to the Perl family but was actually designed from scratch, not as a refactoring of the Perl 5 codebase. Through its two-year-old (released) history, it has increased performance by several orders of magnitude, arriving recently to the point where it can be safely used in production. In this paper, we are going to compare the historical and current performance of Perl 6 in a single problem, OneMax, to those of other interpreted languages; besides, we will also use implicit concurrency and see what kind of performance and scaling can we expect from it.

CCS CONCEPTS

• **Theory of computation** → **Evolutionary algorithms**; • **Computing methodologies** → *Distributed algorithms*;

KEYWORDS

Benchmarking, computer languages, concurrency, evolutionary algorithms, Perl, Perl 6

ACM Reference Format:
Juan-Julián Merelo-Guervós and José-Mario García-Valdez. 2018. Performance improvements of evolutionary algorithms in Perl 6. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205651.3208273

1 INTRODUCTION

Performance has always been a concern in scientific computing. Generally, you will want to use the fastest language available to be able to run your experiments in as little time as possible. However, while implementation matters [13], ease of programming, available libraries and supporting community are sometimes more significant concerns, since in scientific computing the target is to optimize time-to-publish the paper, not only time from pressing *Enter* to obtaining the results, and that includes time to get toe program done itself, as well as process results.

In that sense, interpreted languages such as Python, Perl or JavaScript [2, 3, 6, 9, 11, 12, 15] offer fast prototyping, if not the fastest implementation, which usually belongs to compiled languages such as Haskell or Java [10]. However, as proved in the cited paper, that is not always the case and new languages deserve a chance to be tested, mainly if they offer functionalities that might make the implementation of evolutionary algorithms faster or more straightforward.

Besides, the performance of a language is not a static thing; while some languages are happy enough with the level they achieve and focus on other functionalities, newer languages focus on performance in every new release, offering improvements of several orders of magnitude. This has been the case of Perl 6 [17], a new, concurrent, dynamic and multi-paradigm language that has been in development since 2000 and released in December 2015. Since then, it has had a release cycle of one, or sometimes more, releases every month, with a stable release every four months. While initial tests discouraged us from including its figures in the paper where we benchmarked many languages for evolutionary algorithms [5], the increase in performance has been continuous, as well as the implementation of implicit parallelism features.

This paper is specially focused on benchmarking this language for evolutionary algorithms, with the intention of proposing it as production-ready for scientific computing or evolutionary computation experiments.

The rest of the paper is organized as follows. We will briefly present the state of the art in benchmarking evolutionary algorithms in the next section, followed by the set of experiments used to test the performance in Section 3. Results and charts will be presented in Section 4, and we will close the paper by stating our conclusions.

2 STATE OF THE ART

As a matter of fact, there is very little scientific literature on Perl 6, much less applied to scientific computing. The paper by Audrey Tang [17], one of the early programmers of a Perl 6 compiler in Haskell called Pugs, is one of the few we can find. In fact, the paper where she describes the design of the language has had some influence in language design, including the design of Typed Scheme, a functional language [18].

Its sister language, Perl, has been used in Evolutionary Algorithms for a long time, with an early tool but used for minimizing the performance of a network [1]. Since the publication of the `Algorithm::Evolutionary` library circa 2002 [8, 9] it has been applied to many different problems, including solving the MasterMind puzzle [7]. In fact, its speed processing evolutionary algorithms has made it a great tool for evolving regular expressions via the DiscoverRegex and GenRegex tools [16], and even optimizing the yield of analog integrated circuits [4].

Perl 5 was a convenient and multi-paradigm, if not particularly groundbreaking language. Conceptually, you could program an

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208273

evolutionary algorithm in pretty much the same way you would do it in C or C++, which were at the time much faster languages. The fact that it was used proves that languages for implementing evolutionary algorithms are not chosen purely by their raw speed.

However, speed has to be adequate and not vary in orders of magnitude with respect to other, well-established, language. Even if slower, the trade-off might be interesting if a new language offers new ways of implementing evolutionary algorithms that give you some insight on the inner workings of evolutionary optimization. This why in this paper we will set to measure the speed of Perl 6 and its evolution, in order to prove that it has come the time to consider it as a language for evolutionary optimization given the functional and concurrent facilities it now offers.

3 EXPERIMENTAL SETUP

In this experiment we have used the same operators and data already published in [10] that is, crossover, mutation and one-max or count-ones. We have added the Royal Road function [14], mainly with the objective of comparing Perl and Perl 6 and its parallel facilities.

The functions are well known, and the main objective of these tests was, besides comparing performance side by side, see how this performance scales to big, and a bit unrealistic, chromosome sizes. The way the handling of data structures by particular languages is done makes that, sometimes, the speed of dealing with bigger sizes is faster than with smaller sizes; as a matter of fact, in the above mentioned paper Java achieved its best performance for the biggest chromosome size. We tested several data structures in Perl 6 and finally chose a vector (or array) of booleans as the fastest one. In fact the speed of the benchmark is divided in two parts: speed for randomly generating the vector and speed of actually counting the number of ones. In this case, generating a vector of `Bools` was considerably faster than doing the same with integers, although summing them was almost 4 times as slow. That is why we also test a vector of integers in the experiments we show below. These two operations take two lines in Perl 6, as follows.

```
my $ones = Bool.roll xx $len ;
my $maxones = $ones.sum;
```

These two lines show the advantage of this kind of language; the same operation needs several lines and two loops in most other, non-functional, languages. The first one creates an array of random boolean values, generated with `Bool.roll`; `xx` *multiplies* by the length to yield an array of the desired length. And the second line just uses the `sum` method, which is an standard method for arrays and can also be applied to arrays of booleans. In Perl 6, there are many possible ways of achieving the same, but in fact after several measurements we found this was the fastest, even if initially it was much slower than for other languages. Also, as it can be seen, Perl 6 uses *sigils* for variables, this `$` been applied to most kinds of containers. `Bool` is a standard type, and `my` is a *scope* declaration which can optionally include a type or class declaration. A fuller introduction to the language is outside the scope of this paper, but the interested reader can check the documentation at https://docs.perl6.org for a tutorial or a more thorough explanation of all its features and capabilities.

The benchmark consists in 100,000 repetitions of the operation for sizes that are increased by 2 starting from 16 to, when possible, 32768. All experiments took place in a desktop computer with 8 cores running Ubuntu 14.04.5.

All programs are open source, and included in the same GitHub repository that holds this paper in https://github.com/JJ/perl6eo. Data from the experiments is also freely available in the same place.

4 RESULTS AND ANALYSIS

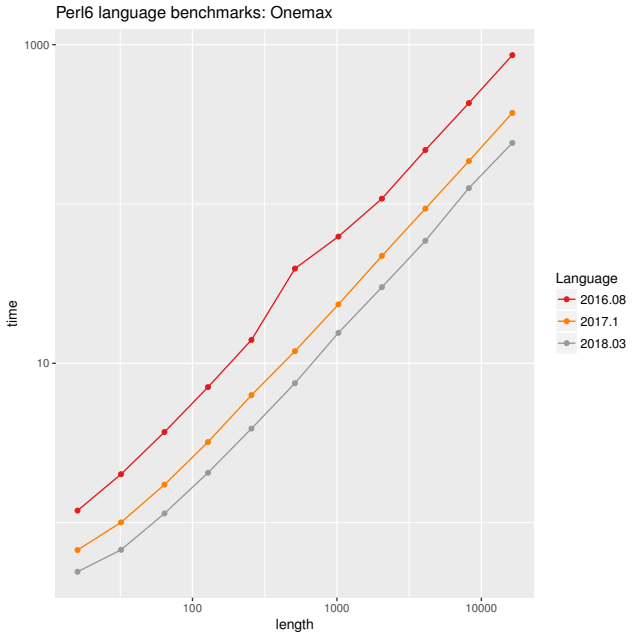

Figure 1: Plot of time needed to perform 100K OneMax evaluations in several versions of Perl 6, from 2016 to the current in 2018. Strings have lengths increasing by a factor of two from 16 to 2^{15}. Please note that x and y both have a logarithmic scale.

The first experiment just measured the speed of the max-ones function across releases of Perl 6; Perl 6 has a monthly release schedule, with version number corresponding to year and month. The result of this operation is shown in Figure 1, and it clearly shows the increase in speed across time, that amounts to almost one order of magnitude from the first version, with a performance that prompted us to exclude it from our initial study, to the current, which is much better.

Despite the improvement, it needs to be compared to the rest of the languages we tested in the previous paper. We have excluded the fastest, mainly compiled, languages, to leave mainly scripting, and some compiled, languages. This comparison is shown in Figure 2. This chart, besides all the measures already published in the previous paper, includes three versions of the one-max in Perl 6. One is the same as above, which uses a boolean representation for the chromosome bits; the second uses an integer representation for the bits and is listed as `IntVector`. This version needed a bit of

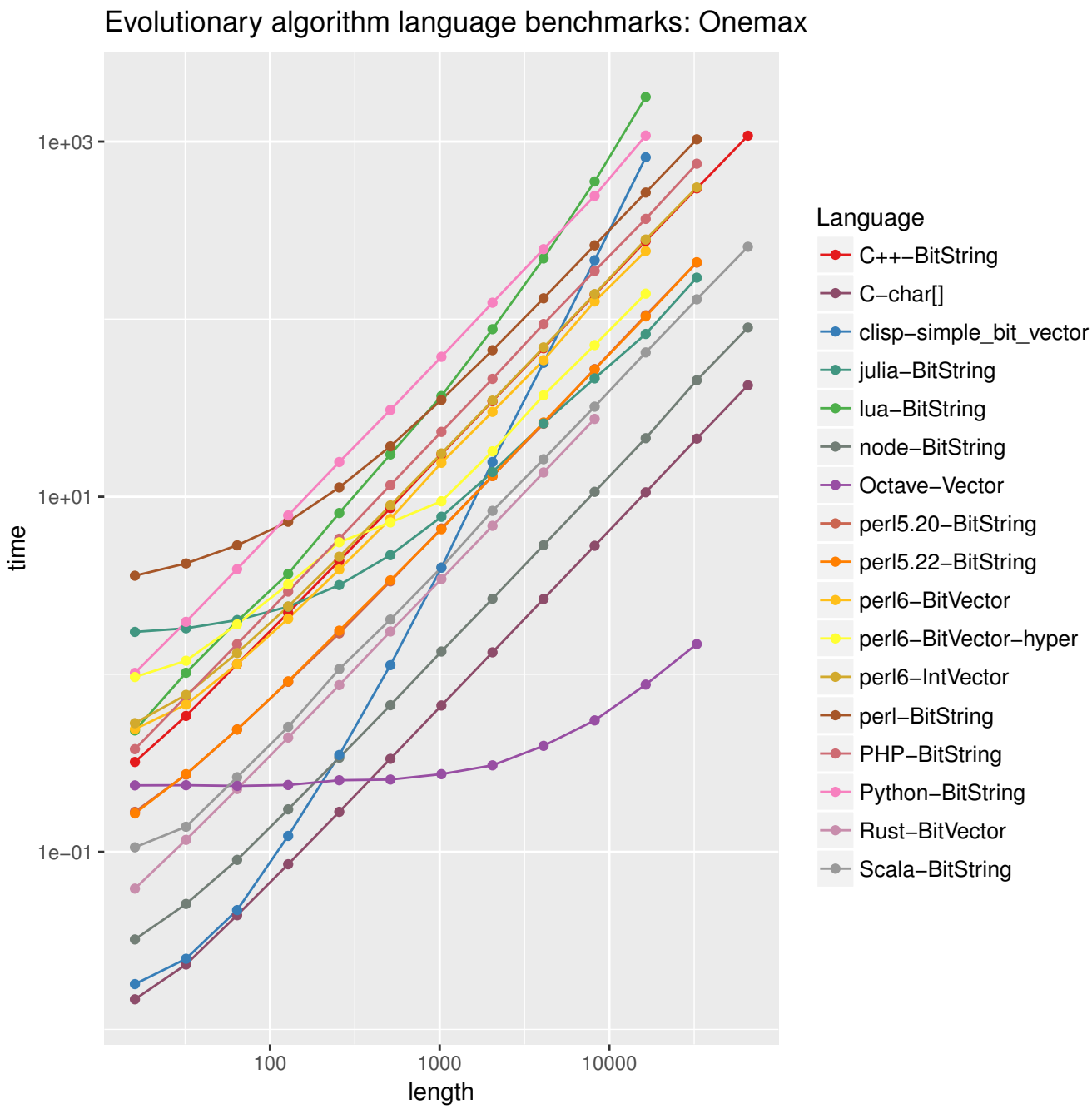

Figure 2: Plot of time needed to perform 100K OneMax function evaluations in strings with lengths increasing by a factor of two from 16 to 2^{15}. Please note that axes x and y both have a logarithmic scale.

hacking which included using a Boolean bit generation and then transforming it to an integer number; however, even that step made it a bit slower than the Boolean version.

The third version, listed as `perl6-BitVector-hyper`, shows one of the unique characteristics of Perl 6: implicit parallelism. The `hyper` and `race` methods, applied to vectors, divide the job into different threads, 4 by default, evaluating different parts of the vector in parallel, without affecting in any way the rest of the operation. In the case above, just changing the line to

```
my $maxones = $ones.race.sum;
```

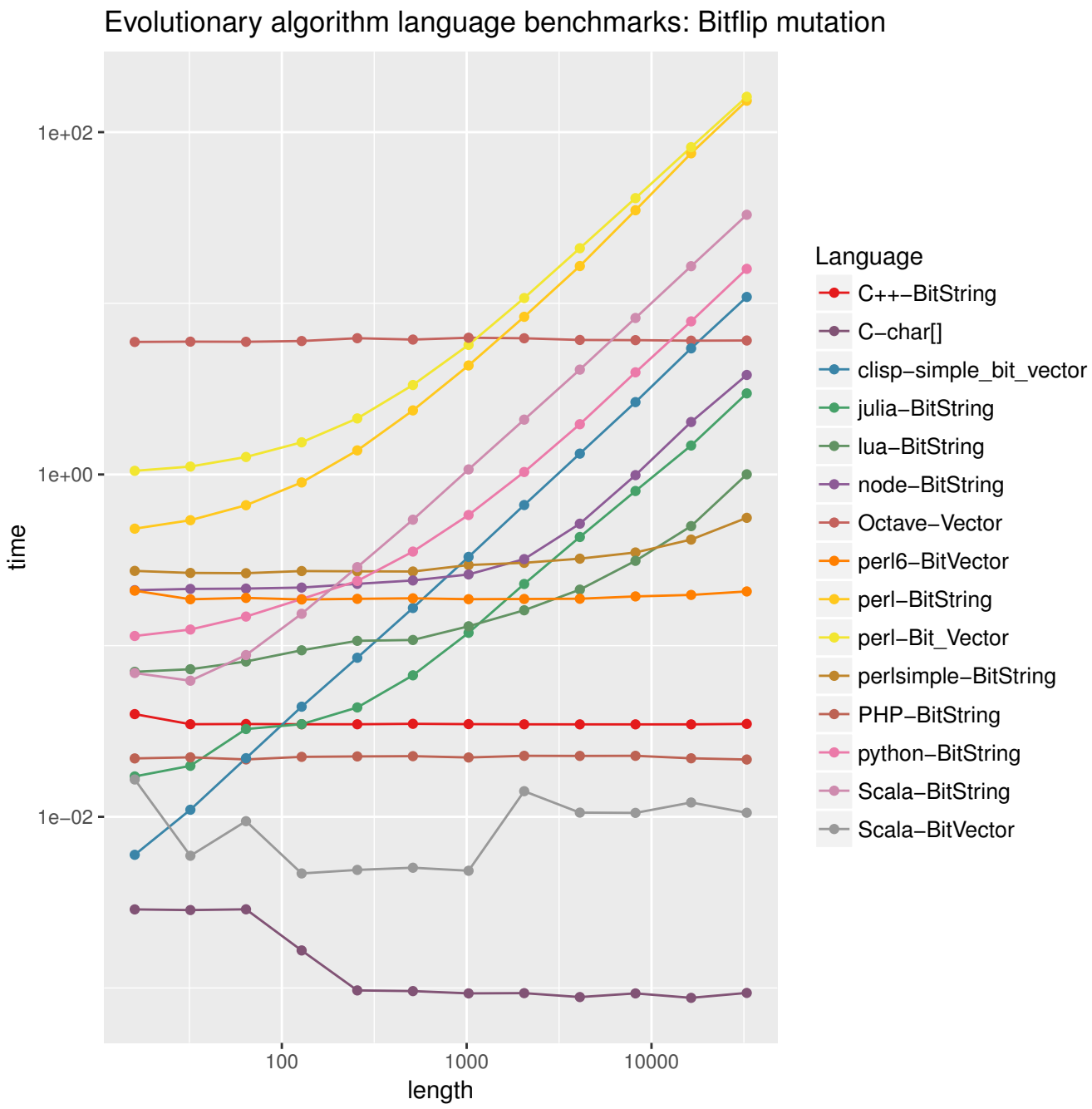

Figure 3: Plot of time needed to perform mutation on 100K chromosomes with increasing lengths from 16 to 2^{15}. Please note that x and y both have a logarithmic scale.

made the sum to be executed in parallel, improving the performance by the number of threads it is using by default. We used `race` instead of `hyper` since the latter forces in-order execution; in our case, the order of the sums is not important and keeping order makes it a bit slower.

The chart shows that, in fact, Perl 6 for this operation is faster, for big sizes, than C++, and overall faster than the Lua language or even Python for a particular representation. For some sizes, it can also be faster than Common Lisp.

In principle, by being faster than more traditional languages, we can prove here that Perl 6 can be not only convenient in terms

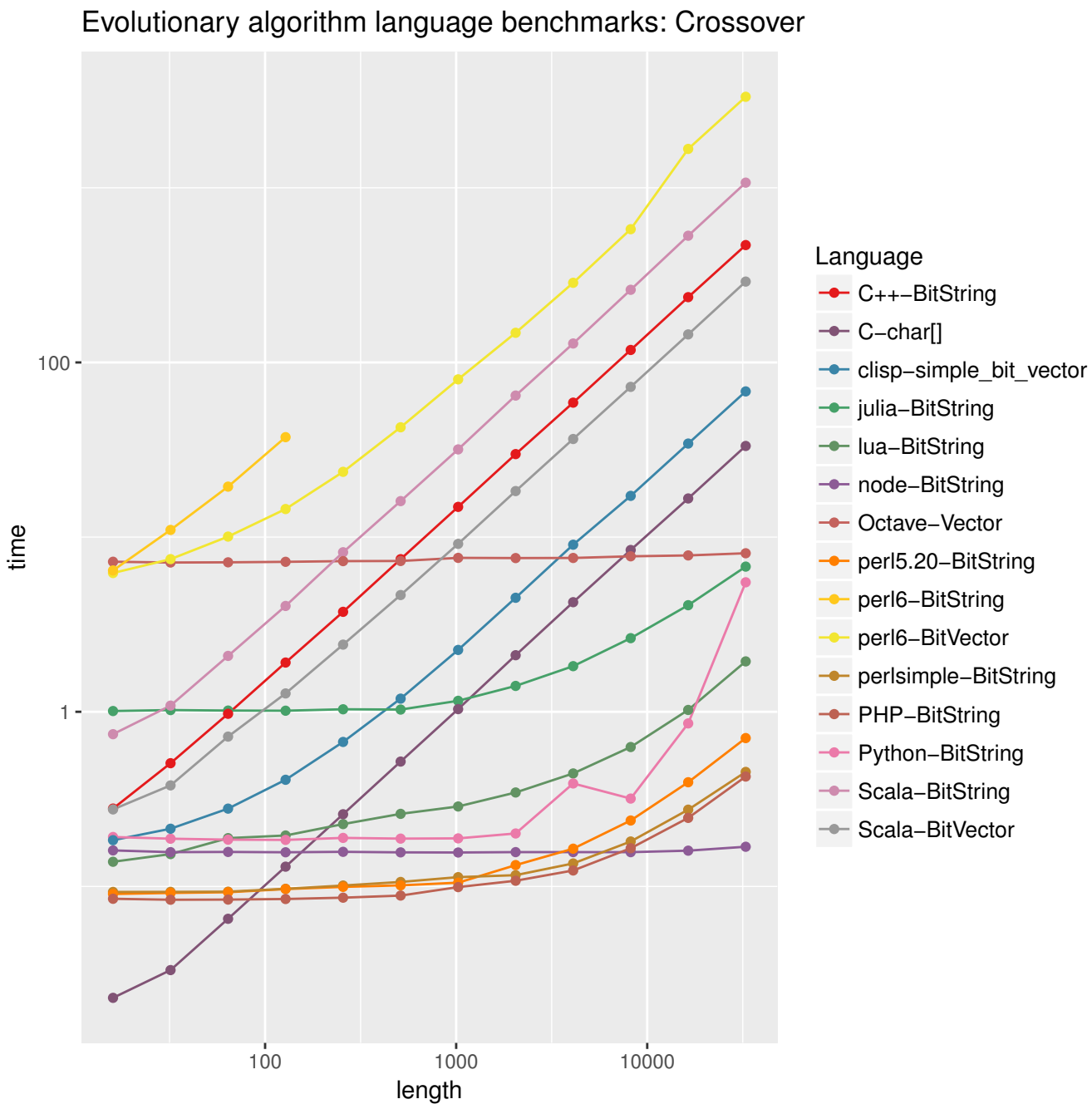

Figure 4: Plot of time needed to perform crossover on 100K chromosomes with increasing lengths from 16 to 2^{15}. Please note that x and y both have a logarithmic scale.

of programming ease (just two lines where other languages need many more lines), but also faster. Let us, however, have a look at the rest of the genetic operations.

The very traditional bitflip mutation comparison chart is shown in Figure 3. The lines used for doing this operation are shown below.

```
my $position = $range.pick;
@ones[$position] = !@ones[$position];
```

In this case we are using `pick` for choosing a random value in a range, which is the chromosome size, and flipping the bit in that random position. Could be done also in a single line, avoiding the `$position` variable; besides, we use the sigil to clearly indicate we

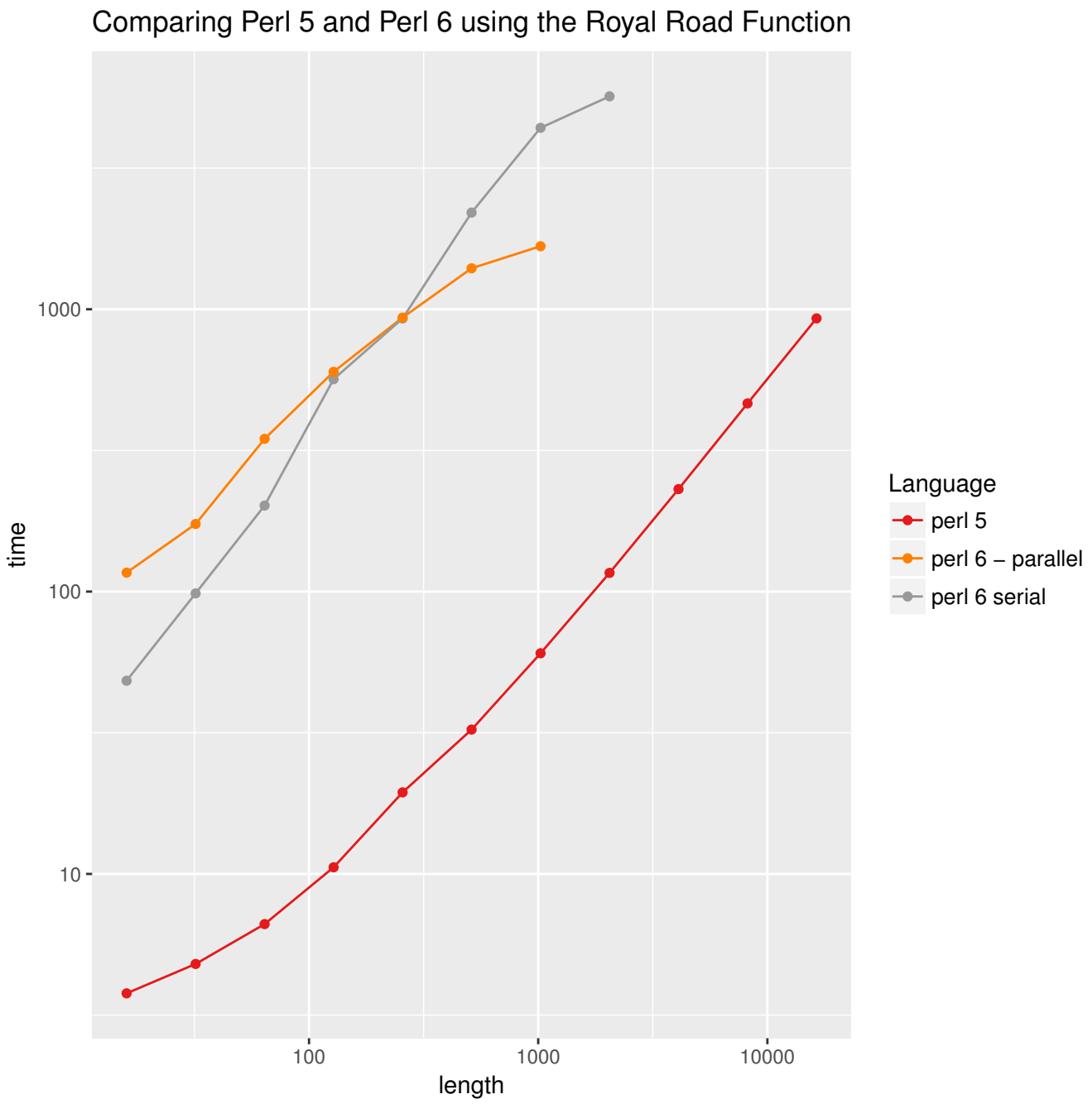

Figure 5: Plot of time needed to perform 100K royal road functions on chromosomes with increasing lengths from 16 to 2^{15}. Please note that x and y both have a logarithmic scale.

are dealing with a vector. We avoided it in the listing above since it made the operation slightly slower.

In this case, Perl 6 is considerably fast, although not the fastest, and the time needed is independent of the chromosome length, a good trait, overall. Once again, it shows a good performance in this operation. Let us examine the next genetic operator, crossover.

The crossover performance comparison chart is shown in Figure 4. In this case, after initial tests, we have gone back again to testing a different representation: a bit string, that is, a string composed of 0s and 1s. Strings have a different internal representation than vectors, and the operations needed are different. While in the first case we could use this line to perform the crossover:

```
@chromosome2.splice($start,$this-len,
    @chromosome1[$start..
      ($start+$this-len)]);
```

, in the second case we used

```
$chromosome2.substr-rw($start,$this-len) =
    $chromosome1.substr($start,$this-len);
```

changing from an array operation to a string operation. And we did so after finding a very disappointing performance, in fact the worst of all languages tested, with the first one. Using a bitstring was not much better, still needing almost double the time of the second-worst language, which is Scala in this case. The fact that these two functional languages have the same disappointing performance, while Scala is usually very fast for all applications, points to the fact that we might be taking the wrong, non-functional, approach to this operation in these languages.

In fact, changing the line to

```
@chromosome2.splice($start,$this-len,
    @chromosome1.skip($start).head($this-len));
```

somewhat improved the performance. In this case we are using functional methods to access different parts of the chromosome. There is around a 20% improvement over the previous line, but still very slow compared to other languages. This proves, anyway, that testing and some help from the community are needed to extract the best performance out of a language; also that idiomatic constructions are in general preferred over generic constructions. It always pays to know the language well.

That is also why we have tested another function, the well known Royal Road, which was proposed as an example of a complicated landscape for evolutionary algorithms. It might also be a complicated performance benchmark. Perl 6 needs a single line to implement this function:

```
my $royal-road= $ones.rotor(4)
    .grep( so (*.all == True|False) ).elems ;
```

In this case, we are using several unique Perl 6 features and doing so in a functional way. For instance, `|` are *Junctions* and `all` becomes `True` or `False` if all the elements in its 4-element block are. That is a very straightforward, and mathematically correct, way to express the Royal Road function. However, it is still slower than Perl by an order of magnitude, as shown in Figure 5. In fact, we had to stop the benchmark, since scaling with size was very bad too.

That is why we used again the `.race` method, which distributes load among threads. Although for smaller sizes the overhead needed to set up the distribution of tasks made it slower, and thus not very convenient for the usual sizes, it became much faster, by almost an order of magnitude, for bigger sizes, proving again that implicit parallelism very conveniently allows to work with big sets of elements, making the result faster. However, it is still very slow. As it becomes the target of optimization in subsequent releases of Perl 6, it will probably improve in speed. The implicit parallel facilities of Perl 6 makes it possible, however, to optimize it at a different level, for instance, population level, which still makes Perl 6 an interesting target for the implementation of evolutionary algorithms. In fact, there are already two implementations available in the Perl 6 module ecosystem, one by the author of this paper, `Algorithm::Evolutionary::Simple`, which includes implementations of the operators shown here. The other one, `Algorithm::Genetic`, makes extensive use of Perl 6 functionalities including roles and `gather/take` loops.

5 CONCLUSIONS

In this paper, we set to prove the readiness of Perl 6, a new programming language, for implementing evolutionary algorithms. Traditionally, these tests have been based purely on performance, to the point that the only questions asked when a new evolutionary algorithm library is released is: "Is it faster than Java/C++?". In this paper we have considered this performance, first historically from the first releases, and then considering the latest releases. Taking into account the improvements in performance experimented along this time, and how seriously performance issues are taken by the developers, we can safely assume that in the medium term Perl 6 will achieve levels of speed comparable with those of other scripting languages, which means that it could be faster than some compiled languages.

On the other hand, a very important consideration is also the facilities that the language offers for the implementation of most classical evolutionary functions. In this case, Perl 6 offers functional methods that allow the chaining of operations, equivalent to function composition, so that in many cases a single line of chained functions is enough to process chromosomes. In many cases, this idiomatic way of doing those operations will result in a faster operation, since idiomatic constructs are usually optimized in every language. In this sense, using either functions or the implicitly parallel methods such as `.race` results in improvements in speed, although for the time being, and in general, Perl 6 is still slower than its sister language, Perl.

Putting both things in the balance, and in general, the conclusion is that the time for implementation of evolutionary algorithms in Perl 6 has arrived, although there is still some way to go in terms of performance. Closely following the development will make the programmer choose the faster alternative for the implementation of evolutionary algorithms, constituting an interesting and promising line of work.

Another line of work will be to use explicit concurrency primitives to implement a concurrent evolutionary algorithms. This is something we will explore in a different paper.

ACKNOWLEDGMENTS

This paper is part of the open science effort at the university of Granada. It has been written using `knitr`, and its source as well as the data used to create it can be downloaded from the GitHub repository https://github.com/JJ/2016-ea-languages-wcci/.

This paper has been supported in part by GeNeura Team, projects TIN2014-56494-C4-3-P (Spanish Ministry of Economy and Competitiveness) and DeepBio (TIN2017-85727-C4-2-P)

We are also deeply grateful to the Perl 6 community, who through the Perl 6 IRC channel and pull requests have helped greatly to improve the code.

REFERENCES

[1] Doina Bucur, Giovanni Iacca, Marco Gaudesi, Giovanni Squillero, and Alberto Tonda. 2016. Optimizing groups of colluding strong attackers in mobile urban communication networks with evolutionary algorithms. *Applied Soft Computing* 40 (2016), 416–426.

[2] Hossam Faris, Ibrahim Aljarah, Seyedali Mirjalili, Pedro A. Castillo, and Juan J. Merelo. 2016. EvoloPy: An Open-source Nature-inspired Optimization Framework in Python. In *Proceedings of the 8th International Joint Conference on Computational Intelligence, IJCCI 2016, Volume 1: ECTA, Porto, Portugal, November 9-11, 2016.*, Juan Julián Merelo Guervós, Fernando Melício, José Manuel Cadenas, António Dourado, Kurosh Madani, António E. Ruano, and Joaquim Filipe (Eds.). SciTePress, 171–177. https://doi.org/10.5220/0006048201710177

[3] Félix-Antoine Fortin, De Rainville, Marc-André Gardner Gardner, Marc Parizeau, Christian Gagné, et al. 2012. DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research* 13, 1 (2012), 2171–2175.

[4] Ivick Guerra-Gomez, Esteban Tlelo-Cuautle, and Luis Gerardo de la Fraga. 2015. OCBA in the yield optimization of analog integrated circuits by evolutionary algorithms. In *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*. IEEE, 1933–1936.

[5] Juan Julián Merelo Guervós, Israel Blancas-Alvarez, Pedro A. Castillo, Gustavo Romero, Pablo García-Sánchez, Víctor M. Rivas, Mario García Valdez, Amaury Hernández-Águila, and Mario Román. 2017. Ranking Programming Languages for Evolutionary Algorithm Operations. In *Applications of Evolutionary Computation - 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Giovanni Squillero and Kevin Sim (Eds.), Vol. 10199. 689–704. https://doi.org/10.1007/978-3-319-55849-3_44

[6] Masatoshi Hidaka, Ken Miura, and Tatsuya Harada. 2017. Development of JavaScript-based deep learning platform and application to distributed training. *arXiv preprint arXiv:1702.01846* (2017).

[7] Javier Maestro-Montojo, Sancho Salcedo-Sanz, and Juan J. Merelo Guervós. 2014. New solver and optimal anticipation strategies design based on evolutionary computation for the game of MasterMind. *Evolutionary Intelligence* 6, 4 (2014), 219–228. https://doi.org/10.1007/s12065-013-0099-6

[8] J. J. Merelo. 2002. Evolutionary Computation in Perl. In *YAPC::Europe::2002*, Münich Perl Mongers (Ed.). 2–22.

[9] Juan-Julián Merelo. 2010. A Perl primer for evolutionary algorithm practitioners. *SIGEVOlution* 4, 4 (2010), 12–19. https://doi.org/10.1145/1810136.1810138

[10] Juan J. Merelo, Pedro A. Castillo, Israel Blancas, Gustavo Romero, Pablo García-Sánchez, Antonio Fernández-Ares, Víctor M. Rivas, and Mario García Valdez. 2016. Benchmarking Languages for Evolutionary Algorithms. In *Applications of Evolutionary Computation - 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 - April 1, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Giovanni Squillero and Paolo Burelli (Eds.), Vol. 9598. Springer, 27–41. https://doi.org/10.1007/978-3-319-31153-1_3

[11] Juan-Julián Merelo-Guervós, Pedro-A. Castillo, and Enrique Alba. 2010. `Algorithm::Evolutionary`, a flexible Perl module for evolutionary computation. *Soft Computing* 14, 10 (2010), 1091–1109. https://doi.org/10.1007/s00500-009-0504-3 Accesible at http://sl.ugr.es/000K.

[12] Juan-Julián Merelo-Guervós, Pedro-A. Castillo-Valdivieso, Antonio Mora-García, Anna Esparcia-Alcázar, and Víctor-Manuel Rivas-Santos. 2014. NodEO, a multi-paradigm distributed evolutionary algorithm platform in JavaScript. In *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014, Companion Material Proceedings*, Dirk V. Arnold and Enrique Alba (Eds.). ACM, 1155–1162. https://doi.org/10.1145/2598394.2605688

[13] Juan-Julián Merelo-Guervós, Gustavo Romero, Maribel García-Arenas, Pedro A. Castillo, Antonio-Miguel Mora, and Juan-Luís Jiménez-Laredo. 2011. Implementation Matters: Programming Best Practices for Evolutionary Algorithms. In *IWANN (2) (Lecture Notes in Computer Science)*, Joan Cabestany, Ignacio Rojas, and Gonzalo Joya Caparrós (Eds.), Vol. 6692. Springer, 333–340.

[14] Melanie Mitchell, Stephanie Forrest, and John H Holland. 1992. The royal road for genetic algorithms: Fitness landscapes and GA performance. In *Proceedings of the first european conference on artificial life*. 245–254.

[15] Víctor M Rivas, Juan Julián Merelo Guervós, Gustavo Romero López, Maribel Arenas-García, and Antonio M Mora. 2014. An Object-Oriented Library in JavaScript to Build Modular and Flexible Cross-Platform Evolutionary Algorithms. In *Applications of Evolutionary Computation*. Springer, 853–862.

[16] David Ruano-Ordás, Florentino Fdez-Riverola, and José R Méndez. 2018. Using evolutionary computation for discovering spam patterns from e-mail samples. *Information Processing & Management* 54, 2 (2018), 303–317.

[17] Audrey Tang. 2007. Perl 6: Reconciling the Irreconcilable. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 1–1. https://doi.org/10.1145/1190216.1190218

[18] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. *ACM SIGPLAN Notices* 43, 1 (2008), 395–406.