Is Julia ready to be adopted by HEP?

Tamás Gál[1,*], *Philippe* Gras[2], *Benedikt* Hegner[3], *Uwe* Hernandez Acosta[4,5], *Stefan* Kluth[6], *Jerry* Ling[7], *Pere* Mato[3], *Alexander* Moreno[8], *Jim* Pivarski[9], *Oliver* Schulz[6], *Graeme* Stewart[3], *Jan* Strube[10], and *Vasil* Vasilev[9]

[1]Erlangen Centre for Astroparticle Physics, Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany
[2]IRFU, CEA, Université Paris-Saclay, Gif-sur-Yvette, France
[3]CERN, European Organization for Nuclear Research, Geneva, Switzerland
[4]Center for Advanced Systems Understanding, Görlitz, Germany
[5]Helmholtz-Zentrum Dresden-Rossendorf, Dresden, Germany
[6]Max-Planck-Institut für Physik, Munich, Germany
[7]Laboratory for Particle Physics and Cosmology, Harvard University, Cambridge, MA, USA
[8]Universidad Antonio Nariño, Ibagué, Colombia
[9]Princeton University, Princeton, NJ, USA
[10]Pacific Northwest National Laboratory, Richland, WA, USA

Abstract. The Julia programming language was created 10 years ago and is now a mature and stable language with a large ecosystem including more than 8,000 third-party packages. It was designed for scientific programming to be a high-level and dynamic language as Python is, while achieving runtime performances comparable to C/C++ or even faster. With this, we ask ourselves if the Julia language and its ecosystem is ready now for its adoption by the High Energy Physics community. We will report on a number of investigations and studies of the Julia language that have been done for various representative HEP applications, ranging from computing intensive initial data processing of experimental data and simulation, to final interactive data analysis and plotting. Aspects of collaborative code development of large software within a HEP experiment has also been investigated: scalability with large development teams, continuous integration and code test, code reuse, language interoperability to enable an adiabatic migration of packages and tools, software installation and distribution, training of the community, benefit from development from industry and academia from other fields.

1 Introduction

High energy physics (HEP) research requires dedicated software, used to run the experiments, perform calculations, make simulations and analyse data. Software development is part of the physicists' activity. We all remember the frustration created by the need to learn computer programming, or for those who had already this skill to learn using HEP-specific software frameworks, while we were eager to focus on physics and obtain results. The youngest had

*Speaker. tamas.gal@fau.de

the chance to use Python because of its ease of use. Nevertheless, they rapidly discovered, that they also need to learn C++ when high computing performance is needed.

The Julia programming language, which was specifically designed to allow researcher to use a single language providing high-performance and easy of programming at the same time, has attracted attention of our community in the last couple of years [1, 2].

Most of us have experienced in their carrier a programming language transition in the HEP community, which were all well motivated. The challenges represented by the current paradigm based on the combination of two languages (mainly Python and C++) and the maturity of Julia which is designed to address the two-language problem may be a sign of motivation for a new transition.

After having looked back at the past language transitions, we will discuss the two-language problem encountered with the current paradigm. We will then present the assets of the Julia programming language and try to answer to the question if Julia is ready to be adopted by the HEP community.

2 Programming language changes in HEP

Transition from Fortran to C++ occurred in the 1990s. It has been motivated by the object-oriented paradigm provided by the new language and the advantages it provides in terms of physics event modelling, software maintainability and extensibility [3–6].

The Python language entered in HEP in the 2010s. Its ease of use, the library ecosystem, in particular for machine learning, the packaging systems, and the interactivity have motivated its adoption. With this transition HEP moved to a multi-language paradigm, where a higher-level language is used for tasks that are not computing intensive or as high level interface to libraries using compiled code (like e.g, ROOT, NumPy and Theano).

The evolution of programming languages used by HEP researchers is studied by counting non-fork GitHub repositories created by people that forked the CMSSW software of the LHC CMS experiment [7]. The results are shown in Fig. 1. For comparison, the same counting is done for persons who forked the NumPy repository, we identify as data scientists, and shown in Fig. 2. This method has its limitations, in particular in terms of the bias coming from the programming language of the probe project (C++ for CMSSW, Python for NumPy). The HEP community seems to follow the data scientist trends, apart for Rust. Python peaked in 2020-2021, while Julia is slowing emerging. Rust shows an increased popularity in the data scientist sample and may take over C++ in the next years.

3 The two-language problem

Introduction of Python brings to HEP the two-language problem [8] encountered in other research fields. The problem is characterised by the use of different languages for easy-and-fast development and for high-performance computing. The approach has two issues, first it requires learning several languages, second it requires frequent port of code initially developed in the easy language to the high-performance one.

For Python, the problem is mitigated by the ease to interface to other languages it provides. Nevertheless, development of high-performance code in Python is demanding. It requires: profound understanding of computer architecture and language interdependence; expertise in the art of programming without loops or more precisely delegating loops to underlying libraries, so called “vectorization”; use of different technologies like Numba, Cython, Pypy to speed up the code. The maintenance overhead rapidly escalates with each additional technology. Moreover, the solutions to overcome with native slowness of Python put constraints on the developer who needs to write its code in a particular manner.

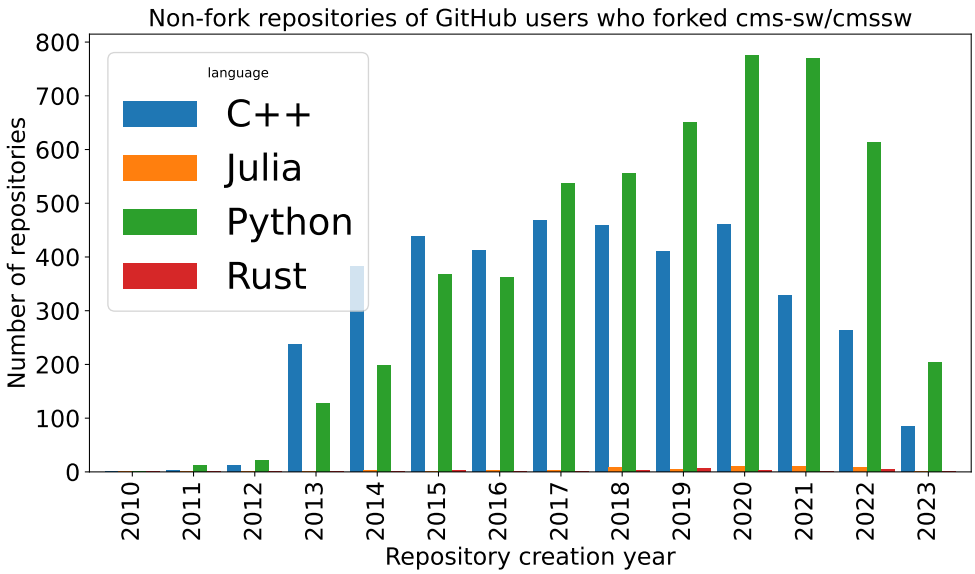

Figure 1. Evolution of the number of repositories per programming language created by users that forked the CMSSW software used as a marker of HEP scientist.

4 The Julia assets

The Julia programming language, which was designed specifically to address the two-language problem, provides in a single language high-performance and ease of programming. The sketch of Fig. 3 shows where Fortran, C++, Python and Julia stand in terms of easy of programming and running speed.

In the comparison performed in Ref. [2] of the running speed for different algorithms implemented in Julia, Python, and C/C++, whose results are reproduced in Fig. 4, we see that Julia provides performance similar to C/C++, while Python can be up to two orders of magnitude slower.

In addition to solving the two-language problems, Julia provides a number of assets compared to C++ and Python. The dynamic multiple dispatch design allows for a massive code reuse and sharing. A detailed comparison of polymorphism mechanisms in C++, Python, and Julia and a discussion of its impact on code reuse can be found in Ref. [2].

Like Python, Julia can be easily interfaced with legacy code written in different languages. The interface with Python is almost transparent and has no overhead. Functions of shared libraries written in C or Fortran can be directly called from Julia with no overhead compared to if the same shared library is interface to a C or Fortran program. Interface to C++ is done using the CxxWrap.jl package. It requires to write some C++ code (called *wrapper*) to define the class and function to bind. The under-development WrapIt package aims to automatise the generation of the C++ code and was used to create a Julia interface, Geant4.jl, to Geant4 [9, 10].

It has a well designed packaging system which is part of the standard library. Use of software package developed by other developers is extremely easy, including packages that uses code developed in another language. For binaries, it includes an infrastructure that builds the code for the different supported architectures to distribute the ready-to-use binaries. Julia natively supports reproducible environment with exact versions of all dependencies. Parallel

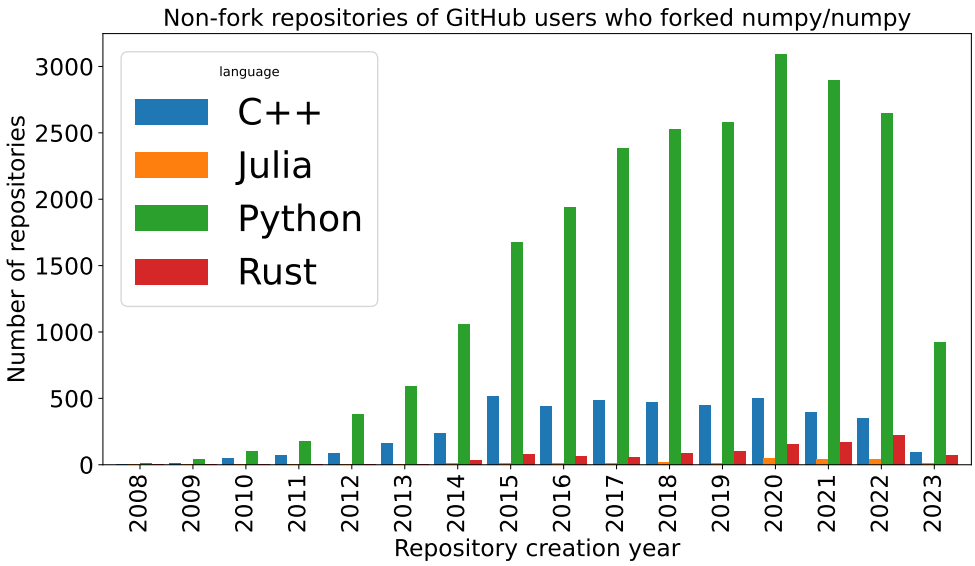

Figure 2. Evolution of the number of repositories per programming language created by users that forked the numpy software used as a marker of data scientist.

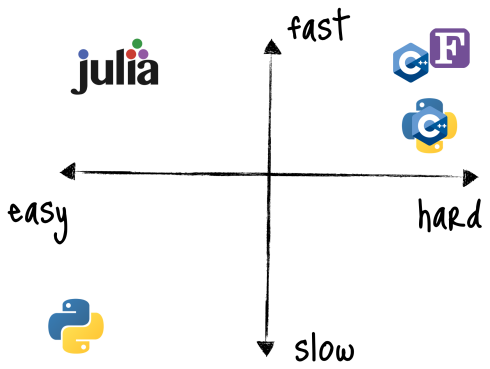

Figure 3. Comparison of different programming languages in terms of ease of programming (abscissa) and running speed (ordinate).

and distributed computing are core features of Julia. A loop can be vectorized, using the simple-instruction-multiple-data (SIMD) CPU support, threads or both. The loop to vectorize is annotated with a single keyword (in practice, a macro) to indicate that the iterations are independent and, in the SIMD case, that arithmetic operations are allowed to be reordered. For the simplest cases, the compiler will find itself that the iterations are independent and will apply SIMD vectorization without the annotation. The LoopVectorization.jl package extends the standard library capabilities to vectorize code, using the same simple-to-use annotation method. Graphic processor units (GPUs) can be used for high-vectorization. The GPU kernels can be written directly in Julia, which makes the use of GPU very easy, both for initial development and for code maintenance. Finally, code can be executed on multiple CPUs in a

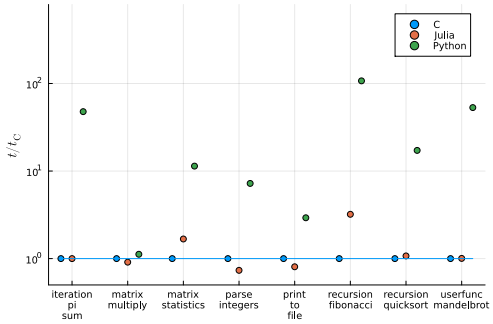

Figure 4. Comparison of C/C++, Python and Julia language performance for a set of short algorithms reproduced [2]. Open BLAS, together with NumPy in the Python case are used for matrix operation. The score is defined as the time to run the algorithm divided by the time to run the C version of the same algorithm.

single or multiple machines in a very convenient way. Batch systems used in the community, as HTCondor are supported.

Several HEP physicists are already using Julia for their research, and packages to read HEP-specific formats have been developed: UpROOT.jl and UnROOT.jl [11] for the ROOT [12] format, LHE.jl for Les Houches Event format [13], LCIO.jl [14] for the Linear Collider Input/Output format. The more general formats Arrow and HDF5, also used in HEP, are supported thanks to the Arrow.jl and HDF5.jl packages.

5 Summary

The venue of Python in HEP computing has revealed the need for a higher-level and easy programming language. Because of the Python's running performance limitations, the language has brought along the two-language problem, experienced before by some other scientific communities that were using higher-level language like R or MATLAB along with a compiled language. Many developments are ongoing to improve the Python performance using extensions or specialized libraries. These solutions are more complex and less flexible for the user that if the language would provide the required performance without need to use extensions or write its code in such a way that it uses the specialized libraries.

Julia is addressing the two-language problem by providing in a single programming language high computing performance and ease of programming. The language is mature and HEP needs are fairly well covered making Julia ready for a large adoption by the HEP community. This language has proven to facilitate in an unprecedented extend code sharing. This can be explained the dynamic multiple dispatch—an intrinsic feature of the language—, the extremely well-designed package management system, and the communication within the Julia open community. It supports distributed and parallel computing and is an excellent language for scientific computing. We believe HEP computing will highly profit in adopting this language as a replacement of the current C++ and Python combination.

References

[1] M. Stanitzki, J. Strube, Comput. Softw. Big Sci. **5**, 10 (2021), https://doi.org/10.1007/s41781-021-00053-3, 2003.11952

[2] J. Eschle et al., Comput. Softw. Big Sci. (2023), https://doi.org/10.1007/s41781-023-00104-x, 2306.03675

[3] D. Boutigny et al. (BaBar), *BaBar technical design report* (1995), http://www-public.slac.stanford.edu/sciDoc/docMeta.aspx?slacPubNumber=slac-r-457

[4] A. Dellacqua et al., *GEANT-4: An Object oriented toolkit for simulation in HEP* (1994), https://cds.cern.ch/record/293084

[5] R. Blankenbecler, L. Lonnblad, Part. World **2**, 130 (1991), http://www-public.slac.stanford.edu/sciDoc/docMeta.aspx?slacPubNumber=SLAC-PUB-5648

[6] L. Lonnblad, A. Nilsson, Comput. Phys. Commun. **71**, 1 (1992), https://doi.org/10.1016/0010-4655(92)90067-9

[7] G.L. Bayatian et al. (CMS), J. Phys. G **34**, 995 (2007), https://doi.org/10.1088/0954-3899/34/6/S01

[8] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V.B. Shah, J. Vitek, L. Zoubritzky, Proc. ACM Program. Lang. **2** (2018), https://doi.org/10.1145/3276490

[9] S. Agostinelli et al. (GEANT4), Nucl. Instrum. Meth. A **506**, 250 (2003), https://doi.org/10.1016/S0168-9002(03)01368-8

[10] J. Allison et al., IEEE Trans. Nucl. Sci. **53**, 270 (2006), https://doi.org/10.1109/TNS.2006.869826

[11] T. Gál, J.J. Ling, N. Amin, Journal of Open Source Software **7**, 4452 (2022), https://doi.org/10.21105/joss.04452

[12] R. Brun, F. Rademakers, Nucl. Instrum. Meth. A **389**, 81 (1997), https://doi.org/10.1016/S0168-9002(97)00048-X. See also https://zenodo.org/record/3895860

[13] J. Alwall, A. Ballestrero, P. Bartalini, S. Belov, E. Boos, A. Buckley, J. Butterworth, L. Dudko, S. Frixione, L. Garren et al., Computer Physics Communications **176**, 300 (2007), https://doi.org/10.1016/j.cpc.2006.11.010

[14] J. Strube, , E. Saba, *jstrube/LCIO.jl: v1.3.0* (2020), https://doi.org/10.5281/zenodo.3666947