Computing and Software for Big Science (2021) 5:10
https://doi.org/10.1007/s41781-021-00053-3

ORIGINAL ARTICLE

Performance of Julia for High Energy Physics Analyses

Marcel Stanitzki[1] · Jan Strube[2,3]

Received: 27 March 2020 / Accepted: 10 February 2021

Abstract
We argue that the Julia programming language is a compelling alternative to implementations in Python and C++ for common data analysis workflows in high energy physics. We compare the speed of implementations of different workflows in Julia with those in Python and C++. Our studies show that the Julia implementations are competitive for tasks that are dominated by computational load rather than data access. For work that is dominated by data access, we demonstrate an application with concurrent file reading and parallel data processing.

Keywords High energy physics · Julia programming language · Data analysis · Multithreading

Introduction

In high energy physics, the programming language of choice for most of the large-scale experiments, like BaBar and Belle, CDF and D0, and the LHC experiments, has been C++. It is used for the simulation and reconstruction chain, as well as for data analysis, which also makes heavy use of the Root [1] data analysis framework. Recently, a change in this paradigm has been observed, where data analysis has been moving towards using Python. Reasons for this move include the faster turn-around of a dynamic scripting language, a lower hurdle for beginners and the growing versatility of Python thanks to packages like NumPy [2] and SciPy [3], developments like the Python–C++ bindings for Root (PyRoot [4]), and data science packages like pandas [5]. Most of the industry-standard deep learning tool kits include Python bindings.

A frequent criticism of Python compared to compiled languages like C++ or Fortran is the performance penalty due to the dynamic character. The Julia programming language [6] originates from the high-performance numerical analysis community and is designed to combine the benefits of dynamic languages like Python with the performance of compiled code by using a just-in-time compiler (JIT) approach. In this paper we demonstrate that Julia exhibits three key features that make it uniquely suitable for data analysis in high energy physics:

1. Julia is fast: We perform benchmarks of data analysis tasks that are typical in high energy physics, implemented in Julia, Python and C++ and compare their performance.
2. Julia is easy to interface with existing C++ libraries: As an example, we have created an interface to the FastJet library [7], the de-facto standard implementation of jet clustering in high energy physics.
3. Julia has support for parallelism and concurrency: We demonstrate how even a trivial implementation of event-level parallelism can accelerate realistic analysis workflows, even if the underlying I/O libraries do not support multi-threading.

Additionally, Julia is interactive, being one of the original languages that integrates with Jupyter [8, 9] notebooks.

The first public release of Julia code was on February 13th, 2013. Within high energy physics (HEP), Julia has been met with interest early on [10, 11], even before the release of 1.0 on August 8, 2018, with its promise of API stability throughout the release 1.x cycle. We argue that adoption by a wider user base was for a while largely hampered

✉ Jan Strube
jan.strube@pnnl.gov

Marcel Stanitzki
marcel.stanitzki@desy.de

[1] DESY, Notkestrasse 85, 22607 Hamburg, Germany

[2] University of Oregon, 1371 E 13th Avenue, Eugene, OR 97403, USA

[3] Pacific Northwest National Laboratory, 902 Battelle Blvd, Richland, WA 99352, USA

Published online: 09 April 2021

by the lack of readers of HEP-specific file formats, such as LCIO and Root. Recently, the number of HEP-specific developments has seen an uptick, with active developments ranging from statistical analysis tools [12], over readers for the Root file format [13, 14], to an interface to data from the Particle Data Group [15]. Julia starts being used to prepare data for HEP publications, e.g. [16],[1] and a toolkit for Bayesian analysis [17] has been implemented in Julia. The main item missing from the eco-system for carrying out a PhD-level physics analysis is an implementation of HEP-specific likelihood fitting codes.

Nevertheless, we argue that the ecosystem is mature enough to support a large variety of tasks encountered in HEP, and here we demonstrate a couple of workflows. All plots in this paper were created using Julia packages.

This paper is organized as follows: first, we give a brief introduction into the features and capabilities of Julia, then we describe the implementation, the setup, and the results of the benchmarks for typical analysis tasks, and then we summarize our results.

The Julia Language

The Julia language is a multi-paradigm, dynamic language with optional typing and garbage collection that achieves good run-time performance by using a just-in-time compiler (JIT). The runtime supports distributed parallel as well as multi-threaded execution. It has been demonstrated to perform at peta-scale on a high-performance computing platform [18], and it has strong support for scientific machine learning [19–22]. The language implementation is open source, available under the MIT license. It is available for download for Windows, MacOS X, Linux x86, FreeBSD platforms, among others [23]. Several universities use the language in their programming courses [24]. As we show in the following, it is also very well suited for developing software in large distributed collaborations like those for high energy physics collider experiments.

Key Features

As Julia is mainly targeted for scientific applications, it supports arbitrary precision integers and floats using the GNU Multiple Precision Arithmetic Library (GMP) [25] and the GNU MPFR Library [26]. Complex numbers and an accurate treatment of rational numbers are also implemented. The base mathematical library is extensive and includes a linear algebra package with access to specialized libraries such as BLAS [27] or MKL [28]. The language fully supports Unicode, which allows the user to write mathematical formulae close to how they would be typeset in a book or paper.

Julia has a builtin package manager, which uses a central repository for registered packages, the so-called Julia registry, but the user has the choice to define their own registry, or to add unregistered packages. Packages with binary dependencies can be built and distributed for a variety of platforms automatically [29]. This frees users from concerns about compiler versions or library incompatibilities.

The three main ways to execute code in Julia are described in the following.

Scripts	Similar to Python, code is written to a text file, which can be executed by the Julia executable.
REPL	The default read-eval-print loop (REPL) in Julia is a highly customizable command line interface, with command history and tab completion. In addition, the REPL comes with several modes, such as a shell mode to execute commands in the shell of the operating system, a help mode to inspect Julia code and a package mode to update and maintain the installed packages. Packages can modify the REPL to make additional modes available.
Jupyter **notebooks**	Jupyter notebooks have become popular in data science applications, where they provide a convenient way to combine code input blocks with graphical output in the same interface. Julia is very well supported in the Jupyter ecosystem.

Implementation Details and Code Distribution

Our study of using Julia in a typical high energy physics analysis workflow uses the LCIO event data model and persistence format, which has been developed since 2003 [30]. The code is released under the BSD-3-Clause license and hosted on Github [31]. The release contains the C++, Fortran, Python, and Go code by default. The generation of Fortran bindings (through C method stubs) can be enabled by a flag at build time. The Python bindings can be enabled at build time, if Root is present. In this case, Root can generate dictionaries for the LCIO classes, which are then made available to Python through the PyRoot mechanism.

[1] Based on private communication with one of the authors.

Additionally, the distribution contains some convenience functions to make the bindings more *pythonic*.

The JULIA bindings for LCIO (`LCIO.jl` [32]) were created by wrapping the C++ classes using the `CxxWrap.jl` JULIA module and the corresponding C++ library [33]. This module requires declaring each class and method to be exposed on the JULIA side to the wrapper generator, which then automatically creates the boilerplate code to translate between JULIA and C++. This translation is transparent to the user, and we have kept the LCIO function names intact, which means that users can still take advantage of the documentation of the upstream library to learn about functionality. The workflow for this is similar to that for creating `Boost.Python` bindings.

This method allows fine-grained control over how the bindings should behave on the JULIA side. For example, it allows us to define a more convenient return type for the three- and four-vectors in LCIO, which are bare pointers in C++, but properly-dimensioned arrays in JULIA. Additionally, the JULIA bindings allow the use of collections of properly typed objects, where LCIO only provides collections of pointers to a base class. This frees the user from having to look up the class type for each of the collections that will be used in the analysis and is helpful particularly for new users. While the shortcomings in this example are straightforward to address on the C++ side, our work shows that this method for defining the bindings is a convenient and powerful way to improve the usability of existing libraries without having to modify the underlying code.

For the deployment, we are using the `BinaryBuilder.jl` [29] package to automatically build binary distributions for the LCIO library, as well as for the C++ component of the wrapper library. The same goes for the FASTJET library and corresponding glue code. Since our package was added to the general JULIA registry, it can simply be installed with the command `add LCIO` in the package manager. This downloads the correct compiled binaries for the package, as well as the glue code between LCIO and JULIA, for the user's operating system, processor architecture and glibc. The user does not need to compile the C++ code or worry about compatibility between different library or compiler versions.

The Test Setup

To demonstrate a simple but realistic workflow, we reconstruct the invariant mass of the process $e^+e^- \to Z^0 \to \mu^+\mu^-$, i.e. production of a Z^0 boson in electron–positron collisions and subsequent decay to a pair of muons. We also perform simple event-shape and jet clustering analyses on a set of $e^+e^- \to Z^0 \to q\bar{q}$ events, i.e. decay of the Z^0 boson to a pair of quark jets. The events were generated with the WHIZARD event generator [34], their interactions with the SID [35] model were simulated using GEANT4 [36–38], and the particles were reconstructed using the particle flow algorithms of the PANDORAPFA package [39]. Events are stored on a dCache system in the LCIO format, which uses zlib for compression. The average file size for the data files is 775 MB; each file contains 32,400 events. To exclude potential OS variations, all results were run on a CentOS Linux 7.7 installation, with 20 Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40 GHz CPUs and 128 GB RAM.

We have implemented the analysis code in C++ and PYTHON, as well as in JULIA and made it available online [40]. All three implementations share the same underlying routines for reading LCIO files. For the computation of the invariant mass in C++ and PYTHON, we use the `TLorentzVector` class of the ROOT package; this is common practice, particularly among students. For simplicity, we are not implementing a vector class with a Minkowski metric in JULIA and instead compute the invariant mass from energy and momentum explicitly.

For our tests the following versions were used:

C++:	gcc 8.3.0
PYTHON:	Python 3.7.6
ROOT:	ROOT 6.22.0
JULIA:	Julia 1.5.0

The ROOT version used by the C++ and PYTHON programs was 6.08/06. The scientific diagrams in this document were created using the histogram (`OnlineStats.jl` and `Plots.jl`) and fitting packages (`Distributions.jl`) from the JULIA General registry [41].

Benchmarking

We compare the performance of different implementations for a number of toy analysis tasks. These tasks are simple but representative workflows that include an I/O-dominated application, an application with a heavy computational component, and an application where the computation is carried out by a third-party library. Finally, we compare implementations of a complete processing chain at different levels of parallelism.

Processing the $e^+e^- \to Z^0 \to \mu^+\mu^-$ Sample

In this simple toy analysis we process a series of LCIO files with $e^+e^- \to Z^0 \to \mu^+\mu^-$ events to reconstruct the Z^0 mass using a Gaussian fit. A real-world application would presumably use a more appropriate shape for the fit, but it shall suffice for the purpose of this study. This is mostly an I/O-dominated application.

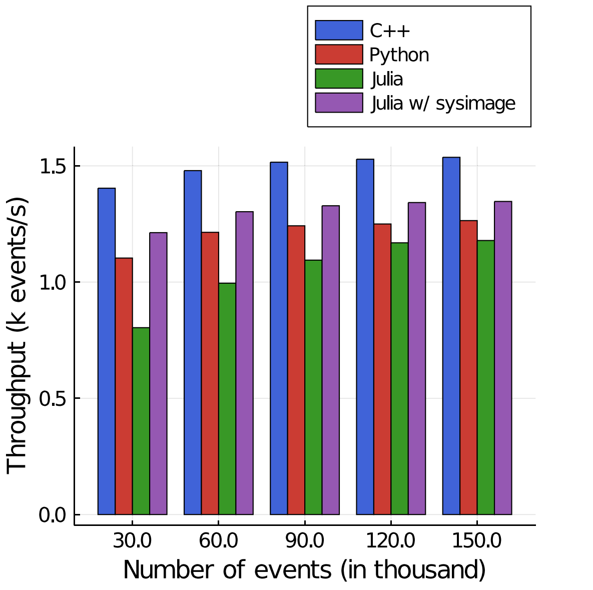

Fig. 1 Event throughput in 1000 events per second for reading different sample sizes and fitting the invariant mass of the Z^0 boson

Table 1 Comparison of the performance of Julia, Python and C++ in processing $e^+e^- \rightarrow Z^0 \rightarrow \mu^+\mu^-$ events

	Julia	Python	C++
Run time (30,000 events) (s)	37.32 ± 1.24	27.19 ± 0.55	21.37 ± 0.18
Average number of events / s	803.88	1103.40	1403.88
Run time (150,000 events) (s)	127.24 ± 4.24	118.62 ± 1.86	97.61 ± 0.34
Average number of events / s	1178.91	1264.50	1536.79
Overhead (s)	15.28	3.90	2.35

The total event samples sizes were 30,000 and 150,000 events, respectively. In order to reduce fluctuations due to external effects like changes in the network or I/O performance, each measurement has been repeated five times. We report here the average and standard deviation of the five runs

For the small-size data set of 30,000 events, the C++ implementation is the clear winner, 75% faster than Julia (see Table 1). Adding the compile time for the first run of the C++ variant of 1.76 s does not change the conclusion significantly. This picture changes when one looks at the larger data sets. The throughputs of the Python and C++ implementations remain more or less constant, as can be expected for interpreted and ahead-of-time compiled languages, respectively. Julia, on the other hand, shows a trend of growing throughput as the data set size increases. This behavior can be attributed to the diminishing contribution of the overhead of the just-in-time compilation step. The trends of processing times for the three language implementations

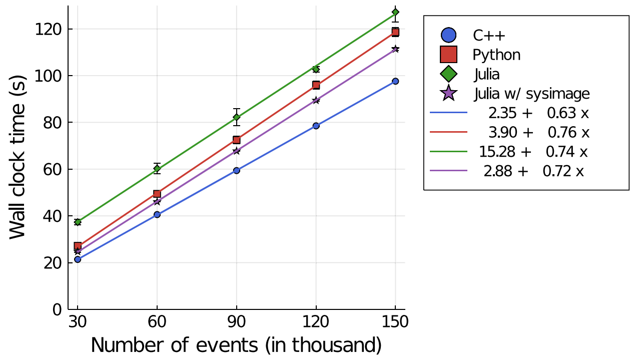

Fig. 2 The dependence of the execution time on the number of events processed for different implementations

are shown in Fig. 1. For comparison, we have compiled the package with the largest start-up time, our `LCIO.jl`, into a static library that the Julia binary links against. This is version is called “Julia w/ sysimage” in our plots. One can see that the much reduced overhead compared to the default version leads to a nearly constant throughput, similar to the C++ and Python versions. We should note, however, that we would expect most users to run the default version with the increased start-up cost.

The question is now how large the overheads of the just-in-time compiled (Julia) and interpreted (Python) languages are compared to the C++ implementation. To get an estimate for this, the execution time (using the Unix `time` command) depending on the number of events processed was measured for the $e^+e^- \rightarrow Z^0 \rightarrow \mu^+\mu^-$ events data set and fit with a linear regression. The overhead in Julia is clearly visible, giving the largest intercept at 15.28 s. This is consistent with time measurements within Julia (that start measuring *after* the initialization step), which are consistently 15 s below what is reported by the Unix `time` command. For Python, we measure an intercept of 3.90 s. The intercept for C++, 2.35 s, can be interpreted as the baseline for the system, and is most likely due to latency of the access to the data files and libraries hosted on different servers. The results are summarized in Table 1 and Fig. 2.

Event Shape Analysis of Hadronic Z^0 Events

While studies of an I/O-dominated process are useful tests of a realistic workflow, the advantage of Julia over Python comes with a more processing-intense algorithm, possibly with multiple nested loops over reconstructed objects. As an example of such an algorithm with a realistic use case, classical event-shape variables like thrust and the Fox–Wolfram Moments [42] were implemented following the implementations in Pythia 6.4 and Pythia 8.2 [43, 44]. For the analysis a set of LCIO files with $e^+e^- \rightarrow Z^0 \rightarrow q\bar{q}$ events was used.

The implementation in Julia (`EventShapes.jl` [45]) is a straightforward translation of the C++ code, which we have verified to produce identical output. This process revealed another important feature of Julia: We found the availability of tools to support the development process to be of a high quality, particularly for identifying the cause for slowdowns and sources of memory allocation. While excellent tools exist to identify the sources of segmentation faults in C++, (e.g. `gdb`), and to track memory allocations and benchmark function calls (e.g. `callgrind` and the `gperf` tool suite), several such tools in Julia are built in and part of the downloaded code, and the interactive nature of the language makes it very easy to isolate pieces of code for benchmarking and debugging. Additional tools are available in third-party packages, and frequently it is sufficient to add macros to the code under investigation to learn more about its run-time performance.

Our first version was several times slower than the C++ version, but a simple annotation with the `@btime` macro from the `Benchmarking` package helped us realize that this was mainly due to memory allocations in the inner loop. Another frequent cause of slowdown is type instability, which prevents the compiler from generating specialized code. In Julia, it is straightforward to identify this symptom with the `@code_warntype` macro. While these features aided us in our code translation, they are even more important in cases where no reference code exists and implementations of algorithms are developed from scratch. The current version of our Julia implementation achieves a processing rate of 287 events per second, while the C++ implementation processes 182 events per second.

Similar to our Julia version, the implementation in Python started out as a straightforward translation of the C++ code. It is understood that this does not result in code that is optimal for performance, but an important aspect of productivity is how close the performance of a first implementation is to optimal. Additionally, it is important that algorithms can be implemented in a straightforward manner from a scientific paper. Vandewalle, Kovacevic and Vetterli [46] define five levels of reproducibility, with the criteria for the highest and most desirable level requiring the research to be re-implemented within 15 min or less.

Our naive translation of the Pythia code to Python processed 2000 events in 61 min and 14 s. To allow a fairer comparison with the Julia code, we have undertaken some efforts in speeding up the Python code as well. Our first attempt was to apply an optimization that is common in NumPy applications, namely vectorization across the outer dimension of the array (i.e., the innermost loop). In our case, however, this did not lead to a measurable improvement in execution speed. Our second attempt was to apply the just-in-time compilation of numba [47]. While the recommended way to annotate the functions did not work for us, due to its inability to ascertain a specific type for some of the Python objects for the compilation step, the `nopython=True` option led to a dramatic speed-up in our tests, processing 204 events per second. We learned from these tests, though, that this comes at the cost of significantly complicating the debugging, because the Python interpreter loses the ability to inspect the numba-annotated sections of the code. Further optimizations are certainly possible, for example by implementing the code in cython [48] or by including tools from the SciPy [3] distribution.

Jet Clustering

We anticipate that most users will use Julia for data analysis in high energy physics in two ways: Either, by writing code directly in Julia, for example, by translating existing codes, as demonstrated in Sect. 5.2, or by calling into existing C++ libraries.[2] As a demonstration of the latter, we have written simple bindings to the frequently used FastJet package (`FastJet.jl` [49]) in Julia, using the same `CxxWrap.jl` package that we used to create the LCIO bindings. Similarly to the LCIO bindings, the code has been added to the Julia registry and can be installed with `add FastJet` in the Julia package manager. Processing the same sample as in Sect. 5.2, we achieved 319 events per second for the Julia implementation and 445 events per second for the C++ implementation.

Parallelizing in Julia

As the number of cores on modern processors keeps growing, the *event-level parallelism* that has been exploited by high energy physics experiments for decades is no longer sufficient to take optimal advantage of the available processing power, mainly due to the memory required to process a single event. Julia has several constructs to support parallel programming. For the purpose of this section, and following the Julia documentation, we distinguish here between *asynchronous*, *distributed*, and *multi-threaded* programming.

In *asynchronous* programming, different pieces of the code run independently of each other, and a scheduler takes care of assigning processing cycles to them. For example, a program could read a file concurrently with, i.e. at the same time as, setting up a canvas for plotting, since the I/O usually has a small ratio of CPU time over wall clock time. This is a straightforward way to speed-up programs that wait for I/O tasks to complete, and this level of concurrency is also available in Python, e.g. in the `asyncio` module. C++

[2] Interacting with libraries written in Python is straightforward in Julia, through the `PyCall.jl` package.

supports this level of programming as *coroutines* in the C++ 2020 standard.

In Julia, the `Distributed` module allows scheduling tasks in different processes, either on the same CPU, or on different CPUs that are connected by a network. In a *distributed* application, different pieces of the code run in different processes. Similar facilities are available in Python (for use on the same node) in the `multiprocessing` module. We are not aware of an implementation in the C++ standard library, but a commonly used library for using distributed computing across different computers is *MPI*, while shared-memory parallelism can be implemented by using programming interfaces like OpenMP and TBB [50].

The level of *multi-threaded* programming that is supported in Julia since version 1.3.0 is not available in Python, due to the global interpreter lock (GIL). In this level of parallelism, different parts of the program run in the same process space. On multiple cores, they can be scheduled such that each thread takes full advantage of a different core. C++ has support for using different threads since the addition of the *threads* library to the `C++11` standard.

Our implementation of multi-threaded event processing is based on Julia`Tasks` that we spawn on different threads, and we have implemented parallel programming concepts on two levels. The communication between different `Tasks` happens via `Channels`, which promotes a design similar to that of the Go programming language, for example. We are combining data readers on separate *distributed* processes (in our case on the same CPU) using the `Distributed` module with data processors running in parallel on different threads using the `Threading` module. Combining these two levels in Julia is straightforward, as shown below. This allows us to make optimal use of the available processing power given the specific I/O and processing characteristics of our application.

File-Parallel Data Access

Many C++ libraries used in high energy physics analyses are not thread-safe. While LCIO has recently gained a thread-safe mode of operation, we are using the conventional, non-thread-safe version of the library to demonstrate a code pattern that allows using such C++ libraries in a parallel program nevertheless. For this, we are taking advantage of the fact that common workflows operate on data sets that frequently span multiple files. In our implementation, we combine several different LCIO readers, each running in its own independent process and reading a different set of files. This comes at the cost of having additional overhead from inter-process communication, but our approach is applicable to basically any C++ library, regardless of whether it is thread-safe or not. Furthermore, it is straightforward to change the number of concurrently running instances of LCIO to optimize the ratio of event readers to event processors. In our implementation, the different LCIO processes run on the same node, but the extension of this kind of concurrency to multiple connected nodes is straightforward. This would allow processing parts of the data set hosted on different machines, but it comes with its own trade-offs of network connectivity, communication between the nodes, and the processing power present on the individual nodes. A detailed investigation of such a use case exceeds the scope of this paper.

Putting It All Together: Processing Event Shapes and Jets in Parallel

Since Julia version 1.3.0, the runtime supports multi-threaded execution, which simplifies the implementation of *within-event parallelism* tremendously. To demonstrate the achievable speed-up, we are processing the event thrust and the Fox–Wolfram moments, implemented in Julia, and the jet clustering in C++ FastJet, for each event. Our example executes multiple event analyzers in parallel on different threads. While this paradigm allows for shared-memory parallelism, we have opted here to also use `Channels`s for communication. In the following we sketch the implementation in Julia, show-casing how straightforward it is to set up multi-threaded event processing using the `Distributed` package. The number of available `workers` can be defined during the Julia startup, e.g. `julia -p` would start eight worker nodes within Julia. However the number of threads needs to be set using the environment variable `JULIA_NUM_THREADS` before or via the `-t` flag when invoking the Julia executable.

In the Julia main function, several `RemoteChannel(()->Channel())` are being opened to ensure communication between the data readers and the data processors. The channels are buffered to a given depth, which allows the readers to continue filling them up to the buffer depth, while the processors deplete them from the other end. A write on a full buffer blocks until at least one element has been removed from the other end. The code skeleton of the `main` function for this is shown in Listing 1, illustrating the usage of channels and spawning the data readers on specific processes (using`@spawnat`, and the spawning of the data processors on different threads within the main process using the`Threads.@spawn` command.

Listing 1 The Julia main function used for the parallel event processing

```
function main()
    # let's start with reading up to four files concurrently
    fnames = RemoteChannel(()->Channel{String}(4))

    # let's presume we can buffer up to 200 events.
    # Adjust according to available memory and
    # relative speed of readers and processors.
    events = RemoteChannel(()->Channel(200))

    # the data readers can signal when they are done reading events
    done = RemoteChannel(()->Channel{Int}(4))

    # up to 10 data processors
    nProcessed = RemoteChannel(()->Channel{Int}(10))

    # spawn the data readers
    for w in workers()[1:NREADERS]
        @spawnat w readEvents(fnames, events, done)
    end
    # spawn the data processors, one per thread on this process
    processors = [
        Threads.@spawn processEvents(events, nProcessed)
        for w in 2:Threads.nthreads()
    ]

    # we now have several functions waiting for input on their channels
    # prepare the file names
    for f in ARGS
            put!(fnames, f)
    end
    close(fnames)

    # wait for all readers to be done
    # then we can close the event queue and the processors can finish
    nDone = 0
    nEvents = 0
    while nDone != nworkers()
            nEvents += take!(done)
            nDone += 1
    end
    close(events)

    # wait for the processing to finish
    totalProcessed = 0
    for p in processors
            wait(p)
            totalProcessed += take!(nProcessed)
    end
    close(nProcessed)
    # if all of the channels are closed, the Tasks can finish
    # at this point, we should have: nEvents == totalProcessed
end
```

The data reader function is shown in Listing 2. The `@everywhere` macro instructs Julia to make this function available on all processes. The data reader ingests events from an LCIO file and pushes them into a `RemoteChannel()`.

Listing 2 The data reader function used for the parallel event processing

```
@everywhere function readEvents(fnames, events, done)
    iEvents = 0
    while true
        try
            # take the next file out of a Julia channel
            fn = take!(fnames)
            LCIO.open(fn) do reader
                for evt in reader
                    # write the read Event into the event channel
                    put!(events, collection)
                    iEvents += 1
                end
            end
        catch e
            break
        end
    end
    # write the number of processed events to done channel
    put!(done, iEvents)
end
```

The data processors that have been spawned using `@spawn` are now listening on the `RemoteChannel()` for events being available for processing as shown in Listing 3.

Figure 4 shows the relative speed-up that can be achieved in this way over a single-threaded C++ application.

Listing 3 The data processor function used to demonstrate multi-threaded processing

```
function processEvents(events, nProcessed)
    while true
        try
            collection = take!(events) #receive next event
            h10, h20, h30, h40 = EventShapes.foxWolframMoments(collection)
            #more analysis and clustering
        catch e
            #store number of Events that have been processed
            put!(nProcessed, iEvents)
            break
        end
    end
end
```

The available speed-up using this paradigm is shown in Fig. 3. It shows clearly that the processing is I/O-dominated: More data readers give a better performance, up to a maximum of 15. The maximum value of 20 readers in our test leads to a decrease in performance, most likely due to contention between different processes. Additionally, one can see that more data processors do not necessarily lead to a better overall throughput. The optimal values for numbers of readers and processors, as well as the buffer depth on the `Channel` between the two, depend on the workload and the details of the hardware configuration. A detailed analysis exceeds the scope of this paper, but our chosen paradigm for setting up concurrent processing makes it trivial to evaluate the performance for a given setup and optimize parameters for larger processing jobs.

For reference, a single-threaded application processing the events in JULIA takes 15 min and 2 s and the C++ processing takes 22 min and 25 s.

Summary

For many years, the high energy physics community has spent significant resources on adding interactivity to their software to facilitate plotting and data exploration. Data science packages developed by other fields have realized the same need for interactive data access, and the standard packages in this sector are all accessible from the interpreted language PYTHON. In this paper, we have demonstrated that the JULIA programming language can offer a compelling option

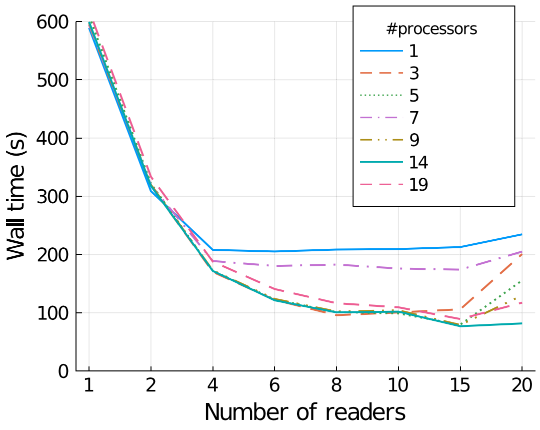

Fig. 3 Time (in s) for processing 233,857 events in 23 files with different numbers of data readers and data processors in Julia

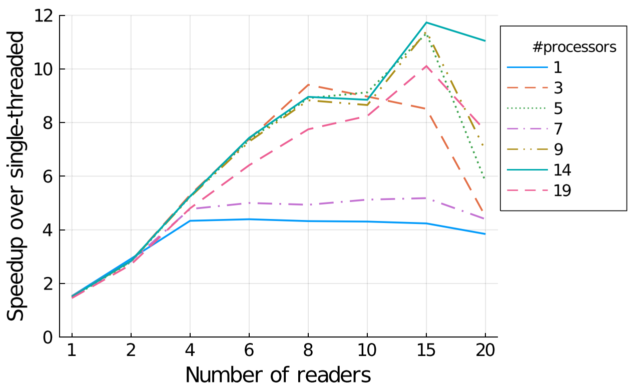

Fig. 4 Relative speed-up of the multi-threaded Julia code relative to the single-threaded application for different numbers of concurrent data readers and data processors. The single-threaded application processes 259 events/s

for data analysis in high energy physics. The interactivity of the language is on par with Python, with support for Jupyter notebooks and a rich ecosystem for plotting and statistical data analysis, as well as deep learning.

With a large set of specialized codes in C++ and Fortran, developed over several decades, interoperability with these languages is a key requirement for any data analysis solution in high energy physics. We have shown that it is straightforward to interface Julia with existing libraries. What is more, the package manager in Julia makes it easy to install the code, without requiring the availability of a compiler for the other languages. The overhead of calling these third-party codes is measurable for simple applications, but negligible for applications with a computation-heavy component.

Fast execution is a requirement for code that has to process millions or even billions of collision events. We have demonstrated that complex algorithms consisting of loops nested multiple levels deep, when implemented in Julia, perform on par with, or even better than, an implementation in C++. The language and package ecosystem have strong support for debugging and studying the software performance. As the language has reached version 1.0 only two years ago, we expect that many opportunities for optimizing the performance further can still be exploited.

The increased use of many-core systems and the growing demand on multi-threading applications to take advantage of the available hardware position Julia extremely well for a growing role in scientific applications in general. We have demonstrated a straightforward way to speed up processing of existing algorithms, without requiring that the algorithms themselves be multi-threaded. The composable parallelism in Julia allows combining this level of parallelism with algorithms that are themselves multi-threaded in nature.[3] A growing set of examples of such algorithms are available online. This feature is what makes Julia an excellent choice as an analysis language in high energy physics, as it allows exploring new algorithms directly at the analysis level without having to drop down to the underlying C++ reconstruction framework for want of better performance.

In summary, Julia currently presents a compelling option for data analysis in high energy physics, and its multi-threading capabilities positions Julia well for future developments. A common challenge for supervisors of summer students is which language to teach them. With C++, significant time is spent teaching them the details of memory management and how to avoid segmentation faults. This is particularly true when interfacing with analysis code in high energy physics, much of which is written in `C++98` style and looks very different from the recommended syntax in `C++17` or newer. With Python, on the other hand, different packages use different syntax to avoid the inherent slowness of `for` loops. Julia advertises itself as solving the two-language problem, where users combine a statically compiled language for performance-critical code with an interpreted language for interactivity. Our studies show that the language keeps this promise and it is straightforward to write high-performance code. The language enables an interactive exploration of data and facilitates the exploration of complex algorithms and is thus an excellent partner to the C++ processing frameworks used in high energy physics.

Acknowledgements The authors gratefully acknowledge the feedback and support of the Julia community, in particular Bart Janssens, the author of `CxxWrap.jl`. We acknowledge DESY (Hamburg, Germany), a member of the Helmholtz Association HGF, for the provision of experimental facilities. Parts of this research were carried out at the National Analysis Facility (NAF). MS acknowledges the support from DESY (Hamburg, Germany), a member of the Helmholtz Association HGF.

[3] For a more detailed discourse on this topic, the interested reader is referred to https://julialang.org/blog/2019/07/multithreading/.

Data Availability Statement "This manuscript has associated data in a data repository. [Authors' comment: The code that was used for the benchmarking in this study has been made available at https://doi.org/10.5281/zenodo.3911414]".

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Antcheva I, Ballintijn M, Bellenot B et al (2009) ROOT—a C++ framework for petabyte data storage, statistical analysis and visualization. Comput Phys Commun 180(12):2499. https://doi.org/10.1016/j.cpc.2009.08.005 ((**40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures**))
2. Oliphant TE (2006) A guide to NumPy, vol 1. Trelgol Publishing, USA
3. Virtanen P, Gommers R, Oliphant TE et al (2020) SciPy 1.0: fundamental algorithms for scientific computing in Python. Nat Methods. https://doi.org/10.1038/s41592-019-0686-2
4. Rademakers F, Canal P, Naumann A et al (2019) ROOT. https://doi.org/10.5281/zenodo.3895860
5. pandas. https://pandas.pydata.org/
6. Bezanson J, Edelman A, Karpinski S, Shah V (2017) Julia: a fresh approach to numerical computing. SIAM Rev 59(1):65. https://doi.org/10.1137/141000671
7. Cacciari M, Salam GP, Soyez G (2012) FastJet user manual. Eur. Phys. J. C 72:1896. https://doi.org/10.1140/epjc/s10052-012-1896-2
8. Kluyver T, Ragan-Kelley B, Pérez F et al (2016) In: Loizides F, Schmidt B (eds) Positioning and power in academic publishing: players, agents and agendas. IOS Press, pp 87–90
9. Jupyter. http://jupyter.org/
10. Pata J (2015) Julia: a fast dynamical language for technical computing and data analysis. https://indico.cern.ch/event/349459/contributions/822791/
11. DIANA-HEP Meeting (2016). https://indico.cern.ch/event/545738/
12. Schulz O, Cornelius G, Hauertmann L, et al (2019) bat/bat.jl. https://doi.org/10.5281/zenodo.3568167
13. Gal T. UnROOT.jl. https://github.com/tamasgal/UnROOT.jl
14. Schulz O, Lusiani A. UpROOT.jl. https://github.com/JuliaHEP/UpROOT.jl
15. Gal T, Schumann J (2020) Corpuscles.jl. https://doi.org/10.5281/zenodo.3933364
16. Tastet JL, Timiryasov I (2020) Dirac vs. Majorana HNLs (and their oscillations) at SHiP. JHEP 04:005. https://doi.org/10.1007/JHEP04(2020)005
17. Schulz O, Beaujean F, Caldwell A, Grunwald C, Hafych V, Kröninger K, La Cagnina S, Röhrig L, Shtembari L (2020) BAT.jl—a Julia-based tool for Bayesian inference
18. Regier J, Pamnany K, Fischer K et al (2018) Cataloging the visible universe through Bayesian inference at petascale. arXiv:1801.10277
19. Rackauckas C, Ma Y, Martensen J et al (2020) Universal differential equations for scientific machine learning. arXiv:2001.04385
20. Innes M, Saba E, Fischer K et al (2018) Fashionable modelling with flux. arXiv:1811.01457
21. Innes M (2018) Flux: elegant machine learning with Julia. J Open Source Sofw. https://doi.org/10.21105/joss.00602
22. Yuret D (2016) Knet: beginning deep learning with 100 lines of Julia. In: Machine learning systems workshop (NIPS). https://goo.gl/zeUBFr
23. Julia—currently supported platforms. https://julialang.org/downloads/#currently_supported_platforms
24. Julia in the classroom. https://julialang.org/learning/classes/
25. Granlund T, Team GD (2015) GNU MP 6.0 multiple precision arithmetic library. Samurai Media Limited, London
26. Fousse L, Hanrot G, Lefèvre V et al (2007) MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans Math Softw 33(2):13-es. https://doi.org/10.1145/1236463.1236468
27. van de Geijn R, Goto K (2011) BLAS (basic linear algebra subprograms). Springer, Boston, pp 157–164. https://doi.org/10.1007/978-0-387-09766-4_84
28. Intel Corp. Math Kernel Library. https://software.intel.com/en-us/mkl
29. BinaryBuilder.jl. https://github.com/JuliaPackaging/BinaryBuilder.jl
30. Gaede F, Behnke T, Graf N, Johnson T (2003) LCIO: a persistency framework for linear collider simulation studies. eConf C0303241, TUKT001
31. Gaede F, Behnke T, Graf N, Johnson T. LCIO. https://github.com/iLCSoft/LCIO
32. Strube J (2020) LCIO.jl: v1.8.0. https://doi.org/10.5281/zenodo.3986687
33. Janssens B. CxxWrap.jl. https://github.com/JuliaInterop/CxxWrap.jl
34. Kilian W, Ohl T, Reuter J (2011) WHIZARD: simulating multi-particle processes at LHC and ILC. Eur Phys J C 71:1742. https://doi.org/10.1140/epjc/s10052-011-1742-y
35. Abramowicz H et al (2013) The international linear collider technical design report—volume 4: detectors
36. Agostinelli S, Allison J, Amako K et al (2003) Geant4—a simulation toolkit. Nucl Instrum Methods Phys Res Sect A Accel Spectrom Detect Assoc Equip 506(3):250. https://doi.org/10.1016/S0168-9002(03)01368-8
37. Allison J, Amako K, Apostolakis J et al (2006) Geant4 developments and applications. IEEE Trans Nucl Sci 53(1):270
38. Allison J, Amako K, Apostolakis J et al (2016) Recent developments in Geant4. Nucl Instrum Methods Phys Res Sect A Accel Spectrom Detect Assoc Equip 835:186. https://doi.org/10.1016/j.nima.2016.06.125
39. Thomson MA (2009) Particle flow calorimetry and the PandoraPFA algorithm. Nucl Instrum Methods Phys Res A 611:25. https://doi.org/10.1016/j.nima.2009.09.009
40. Strube J (2020) Julia_in_HEP_paper. https://doi.org/10.5281/zenodo.3911414
41. Julia Community. The official registry of general Julia packages. https://github.com/JuliaRegistries/General
42. Fox GC, Wolfram S (1978) Observables for the analysis of event shapes in e+ e- annihilation and other processes. Phys Rev Lett 41:1581. https://doi.org/10.1103/PhysRevLett.41.1581
43. Sjöstrand T, Mrenna S, Skands PZ (2006) PYTHIA 6.4 physics and manual. JHEP 05:026. https://doi.org/10.1088/1126-6708/2006/05/026
44. Sjöstrand T, Ask S, Christiansen JR et al (2015) An introduction to PYTHIA 8.2. Comput Phys Commun 191:159. https://doi.org/10.1016/j.cpc.2015.01.024
45. Strube J (2020) EventShapes.jl v0.1.1. https://doi.org/10.5281/zenodo.3698379
46. Vandewalle P, Kovacevic J, Vetterli M (2009) Reproducible research in signal processing. IEEE Signal Process Mag 26(3):37. https://doi.org/10.1109/MSP.2009.932122
47. Anaconda Inc. Numba—A High Performance Python Compiler. http://numba.pydata.org/

48. Behnel S, Bradshaw R, Citro C et al (2011) Cython: the best of both worlds. Comput Sci Eng 13(2):31. https://doi.org/10.1109/MCSE.2010.118
49. Strube J (2020) FastJet.jl. https://doi.org/10.5281/zenodo.3929866
50. Intel Corp. Threading Building Blocks. https://software.intel.com/content/www/us/en/develop/tools/threading-building-blocks.html