Bridging HPC Communities through the Julia Programming Language

Valentin Churavy[1], William F Godoy[2], Carsten Bauer[3], Hendrik Ranocha[4], Michael Schlottke-Lakemper[5,6], Ludovic Räss[7,8], Johannes Blaschke[9], Mosè Giordano[10], Erik Schnetter[11,12,13], Samuel Omlin[14], Jeffrey S. Vetter[2], Alan Edelman[1]

Abstract

The Julia programming language has evolved into a modern alternative to fill existing gaps in scientific computing and data science applications. Julia leverages a unified and coordinated single-language and ecosystem paradigm and has a proven track record of achieving high performance without sacrificing user productivity. These aspects make Julia a viable alternative to high-performance computing's (HPC's) existing and increasingly costly many-body workflow composition strategy in which traditional HPC languages (e.g., Fortran, C, C++) are used for simulations, and higher-level languages (e.g., Python, R, MATLAB) are used for data analysis and interactive computing. Julia's rapid growth in language capabilities, package ecosystem, and community make it a promising universal language for HPC. This paper presents the views of a multidisciplinary group of researchers from academia, government, and industry that advocate for an HPC software development paradigm that emphasizes developer productivity, workflow portability, and low barriers for entry. We believe that the Julia programming language, its ecosystem, and its community provide modern and powerful capabilities that enable this group's objectives. Crucially, we believe that Julia can provide a feasible and less costly approach to programming scientific applications and workflows that target HPC facilities. In this work, we examine the current practice and role of Julia as a common, end-to-end programming model to address major challenges in scientific reproducibility, data-driven AI/machine learning, co-design and workflows, scalability and performance portability in heterogeneous computing, network communication, data management, and community education. As a result, the diversification of current investments to fulfill the needs of the upcoming decade is crucial as more supercomputing centers prepare for the exascale era.

Keywords

High-Performance Computing, HPC, Julia, Programming Language, Workflows, Productivity, Performance Portability

1 Introduction

The Julia programming language (Bezanson et al. 2018) was designed in the last decade to be a novel, high-level, dynamic, and high-performance approach to numerical computing. Julia programs compile as efficient native code for several heterogeneous architectures via the open-source LLVM compiler (Lattner and Adve 2004). The syntax builds upon the success of Fortran for multidimensional arrays and mathematical abstractions (Backus and Heising 1964) and combines with a rich ecosystem that includes high-level interfaces for data structures, analysis, visualization, AI frameworks, and interactive computing. Julia was also designed to address aspects that are typically offloaded to a language ecosystem but are still necessary in the overall scientific discovery process (e.g., reproducibility, packaging, environment portability). Julia also includes a powerful macros system for code instrumentation, interactive computing capabilities, and lightweight interoperability with existing C and Fortran codes—especially highly optimized high-performance computing (HPC) software frameworks and libraries. Julia offers a powerful workflow composition strategy because existing highly optimized HPC frameworks can

[1] Massachussetts Institute of Technology, USA
[2] Oak Ridge National Laboratory, USA
[3] Paderborn Center for Parallel Computing, Paderborn University, Germany
[4] Department of Mathematics, University of Hamburg, Germany
[5] Applied and Computational Mathematics, RWTH Aachen University, Germany
[6] High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Germany
[7] Laboratory of Hydraulics, Hydrology and Glaciology (VAW), ETH Zurich, Switzerland
[8] Swiss Federal Institute for Forest, Snow and Landscape Research (WSL), Birmensdorf, Switzerland
[9] National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, USA
[10] Centre for Advanced Research Computing, University College London, Gower Street, London, WC1E 6BT, United Kingdom
[11] Perimeter Institute, 31 Caroline St. N., Waterloo, ON, Canada N2L 2Y5
[12] Department of Physics and Astronomy, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1
[13] Center for Computation & Technology, Louisiana State University, Baton Rouge, LA 70803, USA
[14] Swiss National Supercomputing Centre (CSCS), ETH Zurich, Switzerland

be combined seamlessly with high-performance Julia kernel code for computation and data management on heterogeneous systems. This creates a powerful synergy for programming HPC systems as more emphasis is placed on performance portability and programmer productivity in the overall workflow process, beyond simulations (Ben-Nun et al. 2020).

Software development that targets HPC facilities for scientific discovery is a nontrivial and highly specialized task (Parashar et al. 1994). Efficient use of HPC facilities for computational science and engineering (CSE) is a multidisciplinary orchestration among several stakeholders. This process requires intimate knowledge of the application's target domain, the targeted system's architecture, and the algorithms in the frameworks and libraries that handle the scalable computation, communication, and data performance aspects within the co-design process. As we reach the physical limits of Moore's Law in semiconductor technology (Moore 1998; Shalf and Leland 2015), several heterogeneous architectures and programming models have emerged (Vetter et al. 2018) during a time in which the first exascale systems are being deployed for the HPC community. On the software technology side, major vendors have converged around the LLVM open-source project (Lattner and Adve 2004) as the back-end technology of choice for their plethora of compilers and programming models. LLVM's modularity, reusability, and platform-agnostic intermediate representation (IR) enables the desired productivity and performance portability characteristics. At the same time, custom hardware accelerators are powering the computational demands associated with AI applications at a wide range of scales. Consequently, the current landscape offers unique opportunities to rethink traditional HPC aspects such as end-to-end co-design for performance portability of complex workflows, large-scale rapid prototyping, and collaboration with dominant cloud and mobile computing ecosystems (Reed et al. 2022).

The present work outlines our view that Julia can challenge the current status quo—in which high-level languages designed with productivity in mind cannot easily achieve the desired levels of performance—while also reducing the costs associated with the learning curve, implementation, and maintenance of an infrastructure based on compiled HPC languages. Much of Fortran's success can be attributed to providing an answer to the original question (Backus 1980): "Can a machine translate a sufficiently rich mathematical language into a sufficiently economical program at a sufficiently low cost to make the whole affair feasible?" Julia attempts to solve a similar technical and economical challenge according to the current landscape by expanding on the traditional HPC focus of simulation performance towards workflow applications. Just like Fortran has been the dominant language for science in the last several decades, Julia can be seen as a unifying domain-specific language (DSL) for science that targets modern HPC requirements for simulations, data analysis, workflows, and interactive computing. The expected return on investment for leveraging Julia

is an increase in productivity when addressing the end-to-end co-design needs of multidisciplinary HPC projects, without a drop performance portability, while also keeping development in a single unifying language and ecosystem. The latter is particularly important in the convergence of AI + HPC workflows for science as AI has been one of the primary drivers in computational sciences in the past decade (Stevens et al. 2020).

The rest of the paper describes what makes the Julia language an attractive investment for scientific discovery with HPC. Section 2 provides background information on the history and efforts around programming languages for HPC, including initiatives that led to the proliferation of current programming models. Section 3 describes the community adoption, interest in leadership facilities around the world, and the package development and deployment process to enable reproducible science at those centers. Section 4 outlines the value of Julia as a first language for teaching HPC concepts. Performance and scalability, which are key aspects of HPC's ethos, are described in Section 5, including experiences in heterogeneous architectures that combine the power of CPUs and GPUs (graphics processing units). Section 6 presents an overview of Julia success stories, including recent research studies that describe performance aspects and community adoption in the broader field of CSE. Section 7 describes the central aspect of Julia's interoperability with C and Fortran that allows access to highly optimized HPC frameworks, along with reusability with Python's existing frameworks, for a powerful workflow composability strategy. Section 8 summarizes our conclusions and vision for Julia and potential opportunities and investments for the HPC community.

2 Background

The development of programming languages for HPC has a rich and varied history. Early on, the needs of HPC and mainstream computing were mostly aligned around number crunching for numerical calculations, which led to the development of Fortran (Backus and Heising 1964) as the first high-level HPC language in the 1950s. To this day, Fortran continues strongly as a leading programming language for HPC owing to its legacy of investments and highly optimized implementations (Kedward et al. 2022). As computing evolved and added more requirements at the system level to perform data movement, parallel processing, analysis, and visualization, C (Kernighan and Ritchie 1988) and C++ (Stroustrup 2013) became the dominant system-level and numerical computing languages in HPC.

At the beginning of the 21st century, the Defense Advanced Research Projects Agency's (DARPA's)

Corresponding author:
Valentin Churavy, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA

    Email: vchuravy@mit.edu

High-Productivity Computing Systems (HPCS) program (Dongarra et al. 2008) described the common practice for HPC software as writing kernels in a compiled sequential language (e.g., Fortran, C, C++) and then parallelizing them in a memory-distributed model based on the standard Message Passing Interface (MPI) (Gropp et al. 1999). HPCS funded an effort to develop new programming languages that targeted productivity, and this resulted in Cray's Chapel Parallel Programming Language (Chamberlain et al. 2007), IBM's X10 (Saraswat et al. 2007), and Sun's Fortress (Allen et al. 2005). Other efforts included those based on Fortran and C extensions, such as Coarray Fortran (Numrich and Reid 1998) and the unified parallel C (El-Ghazawi et al. 2005). In general, these new programming languages offered an alternative to traditional message passing and multithreaded programming models by using approaches such as partitioned global address space (El-Ghazawi et al. 2005; Almasi 2011).

The past decade has seen several disruptive trends that led to the current landscape of extreme heterogeneity: (1) the emergence and adoption of GPU computing as a disruptive technology in HPC (Kindratenko et al. 2009) owing to its performance, programmability, and energy efficiency (Enos et al. 2010); (2) the flattening of Moore's Law in the CMOS technology manufacturing industry; and (3) the adoption of LLVM as the compiler of choice from major vendors. These trends have led to the proliferation of new standardized, vendor-specific, and third-party programming models in the past decade. These models target HPC languages used to manage the increased heterogeneity of contemporary systems: OpenCL (Munshi 2009), CUDA (Buck 2007), HIP (AMD 2008), OpenMP (Dagum and Menon 1998), OpenACC (Wienke et al. 2012), SYCL (Reyes and Lomüller 2016), Kokkos (Carter Edwards et al. 2014), and RAJA (Beckingsale et al. 2019) among others.

Overall, programming languages used in HPC are not specifically designed for science, with Fortran being the exception. This has been a sustainable model owing to vendor and community support, especially for C++ and Python as rapidly evolving general-purpose languages. The HPC software stacks funded by the US Department of Energy's (DOE's) Exascale Computing Project (ECP) (Heroux et al. 2018; Heroux 2019; Dongarra et al. 2011) have continued to build upon the legacy of Fortran, C, and C++, and Python's high-productivity ecosystem has been widely adopted for data analysis, AI, and workflow composition (Straßel et al. 2020). Ousterhout (1998) already observed the split of programming languages into two distinct groups: *implementation* and *scripting*. It was anticipated that scripting language interfaces that glue together the underlying system components would become a dominant model with trade-offs and challenges of its own. A major challenge is the bifurcation of the different communities and the high cost for learning and maintaining multiple technologies and ecosystems. This is even more noticeable in the era of AI because frameworks such as TensorFlow (Abadi et al. 2015), PyTorch (Paszke et al. 2019), JAX (Bradbury et al. 2018), and Firedrake (Bercea et al. 2016) target end users in high-productivity languages. Closing the gaps between HPC's needs and ease of use is a nontrivial effort that adds overheads costs (Zhu et al. 2021; Lavrijsen and Dutta 2016).

Julia was designed to prioritize research and development cycles from idea to performance portability for scientific discovery. Reducing the overhead development costs in this landscape is crucial as future systems become more complex and heterogeneous. The unified language approach builds upon the requirements of the scientific communities that are facing these challenges. In this regard, Julia has attracted domain scientists and practitioners from multiple disciplines to create a community that continues to grow and establish synergistic collaborations. We propose that Julia is a sustainable investment for HPC software projects as future challenges continue to add costs to the scientific discovery objectives that drive and justify the large strategic investments in these systems.

3 Community

The Julia language community is made up of many people working in various scientific and technical domains, and even the original Julia manifesto[*] described the target demographic as including scientific computing, machine learning, data mining, large-scale linear algebra, and distributed and parallel computing. The umbrella term for these domains is *technical computing.*

The original developers of Julia aimed to design an open-source language to tackle problems in technical computing, and from there the community has grown to encompass a wide variety of use cases—from web servers, to databases, to numerical simulations on HPC systems. Although Julia is now recognized as a general-purpose programming language, the early focus on technical computing is still apparent. Common challenges for people working in technical computing are reproducibility and software distribution, and we will discuss these problems in Section 3.1. The rest of this section focuses on the HPC subdemographic of the Julia community (Section 3.2), Julia at the National Energy Research Scientific Computing Center (NERSC) (Section 3.3), and the HPC centers around the world (Section 3.4).

3.1 Package development and reproducibility

Julia was specifically designed to fulfill the Fortran dream of automating the translation of formulas into efficient executable code (Bezanson et al. 2017). Additionally, Julia addresses the two-language problem by closing the gap between developers and users of scientific software. This is achieved with an intuitive language and by providing users with tools to more

---

[*]https://julialang.org/blog/2012/02/why-we-created-julia/, accessed 08-16-2022.

easily follow good, modern programming practices—including documentation, testing, and continuous integration. A recent survey of the packages collected in the General registry showed a strong adoption of these practices: over 95% of packages had tests and ran them with continuous integration services, and almost 90% of packages had documentation (Hanson and Giordano 2021). The adoption of these practices is also made simpler by package templates such as those provided by `PkgTemplates.jl` (de Graaf and contributors 2022).

Building on the experience of other languages, Julia comes with a built-in package manager, `Pkg.jl`, which can install packages and manage package environments similar to the concept of virtual environments in Python. Julia package environments are defined by two text files: `Project.toml` and `Manifest.toml`. `Project.toml` specifies the list of direct dependencies of an environment and their compatibility constraints. `Manifest.toml` captures all direct and indirect dependencies of the environment and uses the appropriate versions of each software module for the present environment. When both files are provided, they fully define a computational environment, and this environment can then be recreated later or on a different machine. We use these features in the reproducibility repository described in this paper (Churavy et al. 2022).

Julia packages are set up as Git repositories that can be hosted on any Git hosting services. Many development tools, including continuous integration tools and online package documentation solutions, are well integrated with GitHub and GitLab, which are the two most popular repository hosting services within the Julia community. All versions of packages recorded in the General registry are automatically duplicated by the servers used by `Pkg.jl` to prevent deleted packages from taking their dependents out with them—an unfortunate scenario that played out with the `left-pad` JavaScript package (Williams 2016).

Julia allows for writing an entire software stack in a single language thanks to its unique combination of ease-of-use and speed. However, Julia users often want to use legacy code already written in other languages, such as C, C++, Fortran, Python, or R. Julia offers the capability to call functions in shared libraries written in C and Fortran and libraries written in any other languages that provide a C-like interface. Third-party packages such as `Clang.jl` (Norton et al. 2022) and `CBinding.jl` (Rutkowski 2022) enable the automatic creation of Julia bindings for C libraries by parsing their header files. Some packages enable other languages to be used directly from within a Julia process, including but not limited to `PyCall.jl` (Johnson and contributors 2022) and `PythonCall.jl` (Rowley 2022) for Python, `RCall.jl` (Lai and contributors 2022) for R, and `MATLAB.jl` (Mohamad and contributors 2022) for MATLAB. `CxxWrap.jl` (Janssens 2022) makes it possible to interface C++ shared libraries by using a static binding generator.

Within the Julia ecosystem, binary libraries and executables are usually managed with `BinaryBuilder.jl` (Saba and contributors 2022).

This framework allows package developers to compile pre-built versions of the binaries for all Julia-supported platforms and then upload them to GitHub. The corresponding and automatically generated packages, called *JLLs*, provide a programmatic interface to call into libraries or run executables. The JLLs are regular Julia packages that, when installed, automatically download the corresponding libraries or executables, thus relieving users from the effort of installing or compiling external libraries themselves. That the JLLs are regular Julia packages also means that they can be recorded in the package environment, thus extending the reproducibility of a computing environment to libraries and programs in other languages. The `BinaryBuilder.jl` framework is usually seen as successful because it provides straightforward handling of external libraries in the general cases. This may cause some friction in HPC settings in which users would like to leverage the system's fine-tuned libraries. However there are mechanisms to override the pre-built libraries provided by JLL packages while still using their programmatic interface.
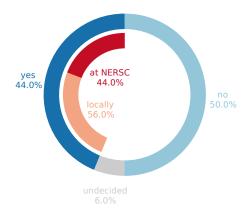
3.2 Uptake of Julia in the HPC community

As Julia places performance at the core of the language, the HPC community has been among the early adopters of the Julia language. Notable examples of early HPC readiness are the petaflop runs at DOE's NERSC (HPCWire 2017). The Celeste Julia code, which analyzes astronomical images, achieved 1.54 petaflops using 1.3 million threads on 9,300 Knights Landing (KNL) nodes of the Cori supercomputer. At the time, this represented an important milestone because experimental and observational science workflows are typically coded using high-productivity interpreted languages that are optimized for rapid prototyping but not for performance. These scientific domains have some of the highest adoption rates for Julia and rely on rapid prototyping, complex workflows, and interactive computing.

3.3 A detailed look at Julia use at NERSC

NERSC is a DOE user facility with approximately 8,000 users. Most users are employed at universities and DOE laboratories, and half are early career scientists, including graduate students and postdocs. Projects using NERSC's HPC systems are funded by DOE program offices: Basic Energy Sciences, High-Energy Physics, Biological and Environmental Research, Fusion Energy Sciences, Nuclear Physics, Advanced Computing Research, and Small Business Innovation Research. Owing to this breadth of research, a survey of NERSC users provides insights into a broad research community.

NERSC monitors the use of the `module load julia` command (among many others) with MODS (Monitoring of Data Systems). MODS captures workflows that use NERSC's official Julia install—users that install their own version of Julia are not tracked. MODS reports that 132 unique, non-staff users loaded a Julia module at least once in 2021. MODS also shows

Figure 1. NERSC user survey: 44% of all respondents (415 NERSC users) plan to use Julia in the future. Of those, 44% plan to use Julia at NERSC.

a gradual increase in Julia module usage at NERSC, but this view is limited. To see a clearer picture of the community's future plans, we surveyed NERSC users and received 415 responses. Most responded within the first 2 days, thereby indicating strong interest. The survey results showed that 44% of respondents are planning to use Julia (Figure 1).

3.4 User support and interest at major HPC centers

Julia is supported by several major HPC centers surveyed in the United States and Europe (see Table 1). Official support at HPC centers takes the form of (1) inclusion of Julia and possibly packages in the official module tree; (2) site-specific configurations (e.g., MPI, I/O); (3) official user documentation; and (4) support for user trouble tickets.

Current support at Oak Ridge National Laboratory's Oak Ridge Leadership Computing Facility (OLCF) (Oak Ridge Leadership Computing Facility) for Summit and Crusher, which is Frontier's test bed system, include recent Julia versions in the user modules. Similarly, the OLCF JupyterHub interface provides custom multithreaded Julia kernels for access to the high-performance file systems. Although user support is available, gaps exist in the official documentation and training (Marques and Barker 2020), and these gaps must be closed to make Julia a viable option for exascale computing.

4 Teaching

Julia's dynamic characteristics and interactive features make it a powerful entry-level tool for teaching, and the official Julia website* offers a selection of online courses. Examples include the Massachusetts Institute of Technology (MIT) modern numerical computing course using Julia for a decade[†]. While ETH Zurich offers a GPU for HPC programming classes using Julia[‡]. The high-level of abstraction enables classroom experiences comparable to Python or MATLAB, and the rich collection of scientific libraries spans a broad

spectrum of applications. As an answer to the two-language problem, Julia can empower domain scientists to dive into HPC development, thereby removing most of the usual barriers that the endeavor would encounter. As such, Julia offers a fast track for domain scientists interested in promoting the development of code on a high level while also offering opportunities for further optimizations, performance engineering, and native tools for precise code analysis.

4.1 Code introspection and performance engineering

In addition to Julia's REPL (read-eval-print loop) component, interactive interfaces such as Jupyter[§] (Jupyter Development Team 2022) and Pluto (van der Plas et al. 2022) provide an engaging learning environment for students with a low barrier to entry. Combined with Julia's high-level syntax, readily available 2D and 3D visualization packages such `Plots.jl` (Christ et al. 2022) and `Makie.jl` (Danisch and Krumbiegel 2021), and a built-in package manager—which also reliably delivers binary dependencies across different operating systems—these frameworks allow one to dive right into the concepts of interest rather than dealing with distracting technicalities or working around missing language features.

At the same time, Julia's just-ahead-of-time compilation delivers fast and pure native code by leveraging the modular LLVM compiler infrastructure. This distinguishes Julia from other dynamic high-level languages, which are typically several orders of magnitude slower, and puts it in the ranks of traditional HPC programming languages (e.g., C, Fortran) in terms of performance and low-level interpretability. As for the latter, the built-in introspection tools, `@code_typed`, `@code_llvm`, and `@code_native`, provide a unique way to interactively explore the compilation of high-level Julia code to intermediate LLVM-IR and low-level machine instructions. In particular, this feature allows one to demonstrate the connection between different variants of code and their respective performance (e.g., owing to the presence or absence of Single Instruction Multiple Data [SIMD] vectorization). Given Julia's competitive speed, students can readily use the language's interactive capabilities to write, analyze, and improve their own domain-specific production codes, thereby making the effort of learning Julia much more profitable for their science.

4.2 Transferable knowledge and experience

Teaching may become a challenging endeavour because it requires the instructor to extract the key concepts from a complex workflow and expose them to students as clear, simple, and concise incremental steps. Conciseness

---

*https://julialang.org/learning/classes/
[†]http://courses.csail.mit.edu/18.337/2018
[‡]https://pde-on-gpu.vaw.ethz.ch
[§]Although Jupyter supports several languages, it derives its name from three programming languages: Julia, Python, and R.

Center Name	System Names	Support Level P U I D				CPU Architecture	Accelerators
Australasia							
NeSI	Mahuika, Māui	✓	✓	✓	✓	Intel Broadwell, Intel Cascade Lake, AMD Milan	NVIDIA P100, NVIDIA P100
Europe							
ARC (UCL)	Myriad, Kathleen, Michael, Young	✓	✓		✓	Various Intel Xeon	Various GPUs
CSC (EuroHPC)	LUMI	✓	✓		✓	AMD Milan	AMD M250X
CSCS	Piz Daint	✓	✓	✓	✓	Intel Broadwell, Intel Haswell	NVIDIA P100
DESY IT	Maxwell	✓		✓	✓	Various AMD Epyc Various Intel Xeon	Various GPUs
HLRS	Hawk	✓	✓	✓	✓	AMD Rome	NVIDIA A100
HPC2N (Umeå)	Kebnekaise	✓	✓		✓	Intel Broadwell, Intel Skylake	NVIDIA K80, NVIDIA V100
IT4I (EuroHPC)	Karolina	✓	✓	✓	✓	AMD Rome	NVIDIA A100
IZUM (EuroHPC)	Vega	✓	✓	✓	✓	AMD Rome	NVIDIA A100
LuxProvide (EuroHPC)	MeluXina	✓		✓	✓	AMD Rome	NVIDIA A100
PC2 (Paderborn)	Noctua 1	✓	✓	✓	✓	Intel Skylake	Various GPUs
PC2 (Paderborn)	Noctua 2	✓	✓	✓	✓	AMD Milan	NVIDIA A100, Xilinx U280, Intel Stratix 10
ULHPC (Luxembourg)	Aion, Iris	✓		✓	✓	AMD Rome, Intel Broadwell, Intel Skylake	NVIDIA V100
ZDV (Mainz)	MOGON II	✓			✓	Intel Broadwell, Intel Skylake	None
ZIB	HLRN-IV	✓	✓		✓	Intel Cascade Lake AP	NVIDIA A100, Intel PVC
North America							
Carnegie Mellon College of Engineering	Arjuna, Hercules	✓	✓	✓	✓	Intel Xeon, AMD Milan	NVIDIA A100, NVIDIA K80
Dartmouth College	Discovery	✓		✓	✓	Various Intel Xeon, AMD Rome	NVIDIA V100
FARSC (Harvard)	Cannon	✓		✓	✓	Intel Cascade Lake	NVIDIA V100, NVIDIA A100
HPC LLNL	Various Systems	✓		✓	✓	Various Processors	Various GPUs
OLCF	Frontier/Crusher	✓	✓	✓		AMD Epyc	AMD MI250X
NERSC	Cori	✓	✓	✓	✓	Intel Haswell, Intel KNL, Intel Skylake	NVIDIA V100
NERSC	Perlmutter	✓	✓	✓	✓	AMD Milan	NVIDIA A100
Open Science Grid		✗	✓		✓	Various Processors	Various GPUs
Perimeter Institute for Theoretical Physics	Symmetry	✓	✓	✓	✗	AMD Epyc, Intel Xeon,	NVIDIA A100
Pittsburgh Supercomputing Center	Bridges-2	✓	✓	✓	✓	AMD Epyc, Intel Xeon,	NVIDIA V100
Princeton University	Several (including Tiger)	✓	✓	✓	✓	Intel Skylake, Intel Broadwell	NVIDIA P100

Table 1. August 8, 2022 snapshot of the Julia support level at different HPC centers (current list is available at https://github.com/hlrs-tasc/julia-on-hpc-systems). User support legend: P = official version preinstalled, U = center provides user support (e.g., center staff answers user questions), I = support for interactive workflows, and D = center provides documentation.

Prepared using sagej.cls

is crucial there because reducing complexity and new concepts to the strict minimum usually accounts for enhanced focus, which in turn enables a steeper learning curve. Teaching is mostly about introducing, exemplifying, and exercising new concepts. Julia's conciseness, performance, and interactive features enable the instructor to go through all these steps with a single code. Julia's high-level syntax permits the instructor to efficiently prototype new concepts into code, and that code actually executes with optimal performance. This is important when teaching algorithmic concepts because users/students usually do not like to wait for their algorithm to complete.

However, the story is dramatically different for HPC. In HPC, one would ideally have some simple high-level code snippets that demonstrate performance-oriented, often parallel and accelerator-based implementations with strong focus on run-time (or implementation) performance. High-level or interpreted languages will mostly fail at this stage because the algorithm design will remain conceptual or require a low-level implementation to fulfil the performance expectations, thereby introducing a significant barrier in the teaching workflow owing to the inherent complexity overhead. The same challenges apply when targeting accelerators such as GPUs. It may be possible to conceptually design GPU kernels in any language; however, when it comes to testing the actual implementation in terms of performance, one would obviously need to have a GPU-compatible code. Julia overcomes the two-language barrier as it allows a single high-level and concise code to be regrouped as the essence of the algorithm or implementation of interest and will most likely enable a high-performance execution of it— be it for demonstration or production purposes. The SAXPY code (Figure 2) exemplifies this by achieving a memory throughput of ∼1,260 GB/s for a high-level broadcasting implementation and ∼1,350 GB/s for compact CUDA kernel and CUBLAS variants on an NVIDIA A100 SXM4 GPU.

Ultimately, students and users can learn about and experiment with basic and advanced HPC concepts within the same interactive language in a portable way. Teaching material can be prototyped on personal computers or laptops, and the same codes can be later deployed on GPUs or HPC servers without code duplication or explicit porting between languages. Moreover, Julia provides a single language to enable experimenting with HPC that can be readily deployed in domain sciences.

5 Scalability and portability

The ability to efficiently deploy a single HPC code on different architectures and at different scales is a key feature for productivity in scientific HPC. Julia offers features that help reduce the complexity of this task, including multiple dispatch, cost-less, high-level abstractions and extensive metaprogramming

```julia
using CUDA
const dim = 100_000_000
const a = 3.1415

x = CUDA.ones(dim)
y = CUDA.ones(dim)
z = CUDA.zeros(dim)

# (a) SAXPY via high-level broadcasting
CUDA.@sync z .= a .* x .+ y

# (b) SAXPY via CUBLAS
CUDA.@sync CUBLAS.axpy!(dim, a, x, y)

# (c) SAXPY via CUDA kernel
function saxpy_gpu_kernel!(z, a, x, y)
    i = (blockIdx().x - 1) * blockDim().x +
        threadIdx().x
    if i <= length(z)
        @inbounds z[i] = a * x[i] + y[i]
    end
    return nothing
end

# launch configuration
nthreads = 1024
nblocks = cld(dim, nthreads)

# execute the kernel
CUDA.@sync @cuda(
    threads = nthreads,
    blocks = nblocks,
    saxpy_gpu_kernel!(z, a, x, y)
)
```

Figure 2. Three different SAXPY implementations based on CUDA.jl (Besard et al. 2018) for NVIDIA GPUs: (a) high-level variant that utilizes broadcasting and array abstractions, (b) simple call into the cuBLAS vendor library, and (c) custom SAXPY CUDA kernel written in and launched from Julia.

capabilities. As a result, powerful low- and high-level packages for performance-portable shared and distributed parallelization have emerged.

5.1 Performance scalability

Julia's base multithreading support and generic high-level packages (e.g., LoopVectorization.jl (Elrod and Lilly 2019), SIMD.jl (Schnetter and contributors 2016)) enable straightforward intranode CPU parallelization. Packages such as CUDA.jl (Besard et al. 2019), AMDGPU.jl (Samaroo et al. 2013), and OneAPI.jl (Besard and other contributors 2020) provide the ability to run Julia code natively on GPUs. Various domain- and method-specific packages (e.g., ParallelStencil.jl (Omlin and Räss 2019), Flux.jl (Innes et al. 2018; Innes 2018)) simplify efficient shared-memory parallelization on GPUs and CPUs for the targeted applications and make it accessible to domain scientists.

Julia includes a generic approach to distributed computing via the Distributed.jl module. A convenient and zero-overhead wrapper for MPI is also available via the MPI.jl package (Byrne et al. 2021). MPI.jl supports CUDA- and ROCm-aware MPI and enables packages that build on it to leverage remote direct memory access (RDMA). Similarly,
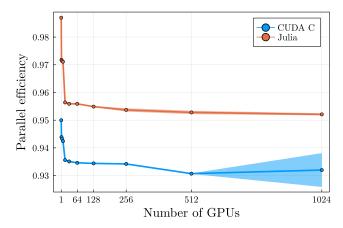
`MPI.jl` enables wrappers for MPI-based libraries for scalable parallel I/O such as: `HDF5.jl`* (Byna et al. 2017; The HDF Group 2000-2010), and the more streaming oriented `ADIOS2.jl`† (Godoy et al. 2020) for data storage and streaming at scale. As for shared memory parallelization, high-level packages can render distributed parallelization simple and efficient for certain classes of applications. Examples include `ImplicitGlobalGrid.jl` (Omlin et al. 2019), which builds on `MPI.jl` and renders efficient RDMA-enabled distributed parallelization of stencil-based GPU and CPU applications on a regular staggered grid almost trivial, and `DistributedArrays.jl` (Contributors 2015), which is a global-array interface that relies on the `Distributed.jl` module.

By combining high-level Julia packages for shared and distributed computing (e.g., `ParallelStencil.jl`, `ImplicitGlobalGrid.jl`), a single high-level HPC code can be readily deployed on a single CPU core or on thousands of CPUs or GPUs. The weak scaling of a Julia-based, coupled, hydro-mechanical 3D multiphysics solver achieves a parallel efficiency of more than 95% on 1–1,024 NVIDIA Tesla P100 GPUs on the Piz Daint Cray XC50 supercomputer at the Swiss National Supercomputing Centre (Figure 3, adapted from Omlin et al. (2020)). These results were confirmed recently by close-to-ideal weak scaling achievements on up to 2,197 P100 GPUs (Räss et al. 2022). The solver was written in CUDA C using MPI (blue data) and translated to Julia (red data) by using `ParallelStencil.jl` and `ImplicitGlobalGrid.jl`. On a single node, the Julia solver achieved 90% of the CUDA C solver's performance (after the initial direct translation) without extensive Julia language–specific optimizations. It should be noted that we apply a strict definition of parallel efficiency, in which the reference performance for one GPU is given by the best known serial implementation in CUDA C and Julia. As a result, the reported parallel efficiency for one GPU is below 100%, and this accounts for the performance loss caused by splitting boundary and inner-point calculations to enable communication/computation overlap (see Räss et al. (2019) for details). This performance loss was more significant for the CUDA C experiments than for the Julia experiments because less-refined parameters were used for the definition of the computation splitting. Thus, the results obtained with CUDA C could certainly be improved by redoing the experiments with better-suited parameters.

5.2 Performance portability

Julia's performance portability story unfolds along several main threads. First, Julia is capable of retargeting the language at a low-level for diverse platforms and accelerators. Second, library writers can use Julia's capabilities to build powerful abstractions. Last but not least, a common array abstraction allows for high-level performance-portable codes.

At the core of Julia's infrastructure sits a flexible and extensible compiler design and a multiple-dispatch

Figure 3. Parallel efficiency of a weak-scaling benchmark using 1 to 1,024 NVIDIA P100 GPUs on the Piz Daint Cray XC50. The blue and orange surfaces visualize the 95% confidence interval of the reported medians. Adapted from Omlin et al. (2020). The raw data and plotting script are available in the reproducibility repository (Churavy et al. 2022).

language feature that enables code specialization for a given run-time type.

Array abstractions. Julia provides powerful array abstractions (Bezanson et al. 2017) that when combined with several implementations allow the user to efficiently express concepts in linear algebra, access optimized implementations, and retarget their programs. At the core of the Julia standard library lies a common super-type, `AbstractArray{T,N}`, for arrays with element type `T` and `N` dimensions. Many subtypes exist: the dense array type `Array{T,N}` (the most commonly used storage type for arrays allocated on the CPU), `Tridiagonal{T}`, `Transpose{T,<:AbstractArray{T,N}}` (a behavioral wrapper that transforms `A[i,j]` into `A[j,i]`), `SparseMatrixCSC{T}`, and `CUDA.CuArray{T,N}` (for arrays on NVIDIA GPUs). The `LinearOperators.jl` (Orban et al. 2020) and `LinearMaps.jl` (Karrasch et al. 2022) packages also provide types that implement linear operators specified as functions without storing any elements (i.e., matrix shell).

All subtypes of `AbstractArray{T,N}` implement an `N`-dimensional array with element type `T`. The way in which elements are stored, which elements are stored, and how the various operations (e.g., addition, multiplication, element access, iteration) are used is left to the implementation. Typically, code that uses arrays (e.g., vectors, matrices, tensors) does not choose a particular implementation but works with any array type. This leads to the same freedom that Kokkos provides—storage and iteration implementation details are decoupled from the algorithms that use these arrays (as much as possible). New hardware back ends for accelerators can be supported in a straightforward

---

*https://github.com/JuliaIO/HDF5.jl

†https://github.com/eschnett/ADIOS2.jl

```julia
using LinearAlgebra
loss(w,b,x,y) = sum(abs2, y - (w*x .+ b)) / size(y,2)
loss∇w(w, b, x, y) = ...
lossdb(w, b, x, y) = ...
function train(w, b, x, y ; lr=0.1)
    w -= lmul!(lr, loss∇w(w, b, x, y))
    b -= lr * lossdb(w, b, x, y)
    return w, b
end
n = 100; p = 10
x = randn(n, p)'
y = sum(x[1:5, :]; dims=1) .+ randn(n)' * 0.1
w = 0.0001 * randn(1,p)
b = 0.0
for i in 1:50
    w, b = train(w, b, x, y)
end
```

Figure 4. A neural network training loop that uses Julia's linear algebra routines.

manner by implementing the appropriate array storage types, similar to `CUArray`.

The user can apply high-level abstractions (e.g., `map`, `reduce`, `mapreduce`, broadcasting) as well as linear algebra routines and other numerical computing operations (e.g., Fourier transforms) to solve scientific problems. For example, the code in Figure 4 implements a simple train loop for a neural network. Notably, to execute this code on the GPU, the user does not need to change the code itself—the user only has to move the data to the GPU. One can achieve this by adding `x = CuArray(x)`, `y = CuArray(y)`, and `w = CuArray(w)` before the loop.

These abstractions are all implemented in Julia itself. Most often, they are dispatched to optimized and specialized operations appropriate for the compute device as well as libraries that provide optimized BLAS operations.

Because the implementation is primarily in Julia, an enterprising user can provide a specialized array implementation and leverage the structure in their own problem. We demonstrate such an scenario in Figure 5. The user can create a wrapper array to encode mathematical knowledge into the array type. In this case, the user needs n numbers to represent a matrix that is dense but structured. The user knows a special algorithm for the largest eigenvalue. With the higher-level abstractions, essentially the same code works on a single CPU, in a distributed setting, or on a GPU.

Powerful libraries. One guiding principle in Julia is that *it is Julia all the way down.* Packages are implemented mostly in Julia itself, as is the base language, standard library, and parts of the compiler. Consequently, there is very little *special code.* By special code, we mean things that the base language (i.e., C or C++) can do that one could not instead implement in pure Julia as a package author. Because of this, there are very few cases in which users would need to write an extension in C or C++.

That said, Julia does rely on external libraries to interact with the operating system and hardware, and it

```julia
using LinearAlgebra

# Build a custom array type
struct DMatrix{T, V<:AbstractVector{T}} <:
    AbstractMatrix{T}
    v::V
end

Base.size(A::DMatrix) = length(A.v), length(A.v)
Base.getindex(A::DMatrix,i,j) =
  A.v[i]*(i==j) + A.v[i]*A.v[j]

# Eigensolver for DMatrix
f(A::DMatrix) =
  λ -> 1 + mapreduce(v -> v^2 / (v - λ)  , +, A.v)
f'(A::DMatrix) =
  λ ->      mapreduce(v -> v^2 / (v - λ)^2, +, A.v)

import LinearAlgebra: eigmax
function eigmax(A::DMatrix; tol = eps(2.0))
    x₀ = maximum(A.v) + maximum(A.v)^2
    δ = f(A)(x₀) / f'(A)(x₀)
    while abs(δ) > x₀ * tol
        x₀ -= δ
        δ = f(A)(x₀) / f'(A)(x₀)
    end
    x0
end
```

Figure 5. A user-defined array type that only stores a vector, v, yet presents the full matrix $vv^T + \mathrm{diag}(v)$ to indexing operations. A custom largest-eigenvalue-solver makes efficient use of this structure via multiple dispatch. Adapted from Edelman (2019).

```julia
using Distributed
addprocs(4)
using CUDA
using DistributedArrays

N = 4_000_000
v = randn(N)*0.1
A = DMatrix(v)

# Explicit data-movement
distA = DMatrix(distribute(v))
gpuA = DMatrix(CuArray(v))

# Execute eigmax on the CPU,
# distributed across multiple processes,
# and on a GPU.
eigmax(A)
eigmax(distA)
eigmax(gpuA)
```

Figure 6. Transparent execution of a program in multiple execution domains.

leverages these libraries when standard solutions already exist for common problems.

The combination of Julia's type system, compiler, efficient execution, metaprogramming and staged programming allows library authors to implement powerful libraries that interact with user code and other libraries. As an example, both `KernelAbstractions.jl` and `ParallelStencil.jl` use macros (metaprogramming) to extend the Julia language with new concepts.

The differential equation ecosystem uses higher-level functions and the capability of the Julia compiler

to specialize these higher-level functions on the user-defined function, thereby leading to cross-optimization between the user and the library code.

Compiling code. Starting at a function call, Julia selects and compiles the most specific function signature. First, Julia propagates the argument types through the body of the function by using an abstract interpretation. At this level, in-lining and constant propagation occur. Afterward, a few optimization passes written in Julia optimize the IR, and the optimized function is translated to LLVM-IR. Julia uses LLVM as a single-function optimizer and to perform scheduling optimization (e.g., loop-vectorization). Then, the function is emitted as a binary and linked in-memory using LLVM's ORC just-in-time.

`GPUCompiler.jl` reuses this infrastructure to collect all statically reachable functions into one LLVM module, which is then compiled and uploaded to the accelerators. This approach is shared among the packages that provide support for accelerators and is flexible enough to support new accelerators/compilation targets.

`GPUArrays.jl` provides generic abstractions and implementations of common functionalities on accelerators, and `KernelAbstractions.jl` provides an extension of the Julia language to write GPU kernels that can be retargeted to different accelerators.

5.3 A language for both beginners and experts

Considerable resources must be invested to train a scientist or engineer to make effective use of HPC. This training typically starts with learning how to program in an undergraduate-level class that is not focused on HPC before being exposed to more advanced topics such as parallel programming, GPU programming, or performance optimization. Often, these introductory programming courses start with a language that is somewhat easy to learn, has a simple syntax, good support for interactivity and visualization, and a strong ecosystem with additional packages and learning material (e.g., Python or MATLAB).

However, this path can be problematic when users eventually switch to a high-performance language (e.g., C++, Fortran) to achieve the required performance for scientific or industrial projects that target compute clusters or supercomputers. As noted before, learning a new programming language is not trivial because concepts often do not translate one-to-one from one language to another, and oftentimes the new language's capabilities are not used to the fullest extent (Scholtz and Wiedenbeck 1990; Shrestha et al. 2020).

The Julia programming language has the potential to overcome this division between easy-to-learn and fast-to-execute languages. Its simple base syntax allows novice programmers to quickly grasp basic concepts such as variables, control flow, or data structures with a convenient style that enables the translation of many mathematical formulae directly into code. Because it compiles to native code, Julia provides the efficiency and optimization opportunities required for production-type computations. This means that as users move to more advanced programming concepts and applications, they continuously accumulate and extend their experience with their programming language and do not need to switch between different tools for rapid prototyping or large-scale application programming. Because Julia provides a REPL, a compiler, and a package manager in one combined solution, it further eases the transition of users between their own laptops, a university cluster, or an extreme-scale machine. Tools, packages, and experience can seamlessly move between different systems and applications.

5.4 Workflow portability and reusability

As demonstrated by NERSC's Superfacility Project (Bard et al. 2022), HPC workloads are rapidly expanding beyond the boundaries of a single data center. At present, efforts to develop multisite workflows are driven by the increasing need to integrate HPC into the data analysis pipelines of large experiments. Furthermore, future DOE initiatives (e.g., the AI for science initiative (Stevens et al. 2020)) emphasize the need for cross-facility workflows. These developments are gradually shifting the emphasis from the HPC application, which must be tailored to specific hardware and software environments, to workflows that incorporate many applications and services at multiple data centers.

Previous studies of state-of-the-art cross-site workflows (e.g., Antypas et al. (2021); Giannakou et al. (2021)) provide a rough anatomy of cross-site workflows, which consist of (1) a data movement layer, (2) portable executables, (3) a workflow orchestration engine, and (4) a control layer that coordinates resources across facilities.

As described in Section 5.2, Julia's syntax provides a natural way to abstract away details of the system's hardware. This abstraction method is aided by the many packages that adopt `Preferences.jl`,* which allows HPC center administrators to configure site-specific settings (e.g., MPI). Notably, users do not need to follow a different deployment recipe for each site. Furthermore, the Julia HPC community is active in developing packages such as `MPItrampoline.jl` as well as bindings for Slurm and the Flux resource manager.

6 Julia success stories

We have claimed that Julia is fast and useful for performance-critical programs. This claim is backed up by the microbenchmarks on Julia's website[†] that show that Julia's performance is comparable to compiled languages such as C and Fortran. Here, we corroborate this claim with additional examples that range from low-level code to high-level libraries and interfaces.

---

*https://juliaparallel.org/tutorials/preferences/

[†]https://julialang.org/benchmarks, accessed 09-28-2021.

6.1 Performance of the same algorithms

Julia can generate efficient machine code for low-level BLAS routines (e.g., matrix multiplication), which are used in various scientific workflows, including machine learning, optimization, statistics, and numerical solution of differential equations. Elrod (2021) demonstrated that highly optimized pure Julia packages (e.g., `Octavian.jl`) can be on par with or even faster than established BLAS libraries (e.g., OpenBLAS, Intel MKL) on Intel's CPU hardware (Figure 7). This is expected because Julia can generate similar LLVM-IR representations that could match the performance of the assembly code from these highly optimized libraries.

Figure 7. Benchmark of matrix multiplication using different BLAS libraries on a single Intel Xeon Gold Skylake 6148 CPU. The raw data and plotting script are available in the reproducibility repository (Churavy et al. 2022). Inspired by a similar plot in `Octavian.jl` (Elrod et al. 2022).

Similar results were obtained for discretizations of ordinary differential equations, which are used in biology, chemistry, and pharmacology. Some example benchmarks[*] that compare implementations of the same algorithm (Dormand and Prince 1980) in Fortran[†] and Julia (Rackauckas and Nie 2017) show that the Julia versions are at least comparable to the Fortran codes and are sometimes even more efficient owing to the enhanced in-lining and other optimizations. These results, which show a comparison of the same numerical methods implemented in different programming languages, extend to partial differential equations, hyperbolic conservation laws, and other transport-dominated phenomena used in weather prediction, climate modeling, and aircraft design. Ranocha et al. (2022) compared the performance of the `Trixi.jl` (Schlottke-Lakemper et al. 2021) Julia package with the mature Fortran code FLUXO[‡] to implement the same algorithms for hyperbolic conservation laws. The Julia code was at least as fast as the Fortran code and sometimes up to 2× faster. More recently, Lin and McIntosh-Smith (2021) showed that in benchmarks across several HPC systems equipped with CPUs and GPUs, Julia's performance either matches or is only slightly behind existing parallel programming frameworks coded in C, C++, and Fortran.

6.2 Algorithmic improvements

Further evidence of Julia's performance and strengths is provided by the `Gridap.jl` Julia package (Badia and Verdugo 2020), which can be used for finite element discretizations in structural engineering, heat transfer problems, and incompressible fluid flows. Leveraging Julia's expressiveness and just-in-time compilation, Verdugo and Badia (2021) reported a finite element assembly performance comparable to FENICS (Logg and Wells 2010), which is based on a DSL and code generation via C/C++. Thus, Julia's expressiveness allows one to have a code that is easier to develop and maintain without sacrificing performance. Furthermore, Julia makes it easier to develop new algorithms with direct support for parallelism, thereby enabling significant speedups in applications that benefit from algorithmic improvements (e.g., pharmaceutical development[§]).

6.3 Common interfaces

One of Julia's strengths is the use of common interfaces in libraries enabled by multiple dispatch. For example, the standard array interface is generic and allows the use of CPUs and GPUs (Besard et al. 2019). Furthermore, automatic differentiation and other tasks do not rely on creating a new array type; instead, they can reuse existing functionality. By using generic programming based on these common interfaces in Julia, packages can work together seamlessly without boilerplate glue code (Karpinski 2019). For example, error propagation with `Measurements.jl` can be combined with spatial semi-discretizations from `Trixi.jl` and time integration methods from `OrdinaryDiffEq.jl` for numerical simulations without special glue code. Additionally, the results can be visualized directly with `Plots.jl`.

At a lower level, common interfaces and operator overloading enable automatic differentiation (Revels et al. 2016), speedups provided by using low- and mixed-precision arithmetic on modern hardware (Klöwer et al. 2020), and uncertainty propagation (Giordano 2016).

At a higher level, such common interfaces are useful for algorithms in certain problem classes: solving linear systems,[¶] differential equations (Rackauckas and Nie 2019), mathematical optimization (Legat et al. 2020), and automatic differentiation (Schäfer et al. 2021). Because the optimal choice of a numerical algorithm depends on the problem, providing all algorithms via a unified interface enables users to swap algorithms depending on their needs. There are focused research efforts to organize such open interfaces to allow seamless

---

[*]https://benchmarks.sciml.ai, accessed 09-28-2021.
[†]http://www.unige.ch/~hairer/software.html, accessed 09-28-2021.
[‡]https://gitlab.com/project-fluxo/fluxo, accessed 09-28-2021.
[§]https://juliacomputing.com/case-studies/pfizer, accessed 09-28-2021.
[¶]https://github.com/SciML/LinearSolve.jl, accessed 03-01-2022.

interconnection in scientific computations (e.g., in the Mathematical Research Data Initiative*). Dunning et al. (2017) demonstrated how such common interfaces can be used via an open-source modeling language for optimization in Julia that is competitive with widely used commercial systems and can even outperform other open-source alternatives.

6.4 Julia's adoption in CSE

Given its features and performance, Julia has demonstrated its readiness for the diverse set of applications in the broader CSE field. Furthermore, we see this readiness as an opportunity for HPC. Working well with CSE applications is crucial for the success of Julia in HPC because these applications allow for testing proven technologies and algorithms at different scales with varying levels of support in a broad community. Success stories in different CSE fields include algebraic geometry (Breiding and Timme 2018), astronomy at petascale (Regier et al. 2018), cancer therapies (Pich et al. 2019), computer algebra and number theory (Fieker et al. 2017), electrical engineering (Plietzsch et al. 2022), epidemic modeling (Weitz et al. 2020), high-performance geophysical simulations (Räss et al. 2022), fluid dynamics (Ramadhan et al. 2020; Ranocha et al. 2022), semiconductor theory (Frost 2017), symbolic-numeric computing (Ketcheson and Ranocha 2021; Iravanian et al. 2022; Ma et al. 2021), quantum optics (Krämer et al. 2018), quantum chemistry (Aroeira et al. 2022), quantum physics (Herbst et al. 2021), and many others.

Typically, the performance of these Julia packages is at least comparable to existing frameworks in low-level programming languages. Sometimes Julia's productivity features even enable improved algorithmic development and simpler reuse of existing specialized implementations, thereby leading to speedups compared to established codes. If highly tuned libraries of core routines are already available with a C interface, then they can be easily accessed from Julia. Thus, a gradual transition that incorporates old code bases is also feasible, as described in Section 7.

7 Interoperability and composability with preexisting code

Owing to the large investment in creating, optimizing, and maintaining HPC software infrastructure, developers do not have to throw away or rewrite their Fortran, C, or C++ codes. Interoperability with preexisting codes has been a top priority and is at the heart of Julia's advantage. Furthermore, to be successful in this space, one must reuse the tremendous work from well-established HPC frameworks. Although there is interest in writing BLAS routines in pure Julia (Elrod 2021) (Figure 7), the ability to call existing vendor-optimized BLAS libraries was important to kick-start the language ecosystem. In Section 7.1, we describe how this capability has grown to integrate preexisting HPC codes into Julia. Section 7.1 describes how these codes can be enhanced with new capabilities. Additionally, Section 7.2 describes how

Julia can be used as an implementation language for new algorithms, thus requiring Julia to be embedded into preexisting HPC software.

7.1 Calling existing codes from Julia

HPC workflows are becoming increasingly complex as a result of increasing resource heterogeneity as well as a growing need for HPC in traditionally non-HPC domains. Yet, traditional HPC code bases are written in languages that prioritize bare-metal performance, and this focus results in low productivity when developing workflows. As a result, we need a programming language that can express complex workflows while still making use of existing codes that encapsulate a large amount (often decades) of institutional and domain knowledge. A common example is incorporating simulation codes and solvers into experimental data analysis workflows.

By far the most common approach in HPC has been to adopt Python as the workflow language and develop high-performance kernels in HPC languages. This approach has a problem: the workflow orchestration layer is not optimized for HPC. To illustrate this problem, we compare the round-trip time to call a C function with Pybind11 (Jakob et al. 2017) vs. Julia's native `ccall` interface (see Table 2 for results). The need to convert between Python data types and native C data types can be seen as an increased round-trip time in the Pybind11 benchmark results. Therefore, workflows coordinated by using Python codes will avoid frequent calls to small C functions—instead opting to combine work in monolithic C kernels. Julia does not have this limitation.

Adding new capabilities to preexisting code. Over the last few years, Julia has become a test bed for the development of new techniques in probabilistic programming (Cusumano-Towner et al. 2019; Ge et al. 2018) as well as scientific machine learning (Rackauckas et al. 2020). For these new techniques, the availability of gradients through automatic differentiation has been key. Similarly, the CESMIX project at the MIT is currently building an integrated framework for uncertainty quantification that greatly benefits from the availability of gradients.

Although Julia has emphasized interoperability with codes written in C, C++, or Fortran from the very beginning, there is an open question as to whether these new techniques can be utilized in codes that are a mixture of Julia + x, where x is an HPC application to which one wishes to apply these techniques. The lynchpin for any attempt at this will be the availability of gradients and the integration of those gradients into Julia's automatic-differentiation frameworks.

Enzyme (Moses and Churavy 2020) and its `Enzyme.jl` Julia front end are an automatic differentiation framework that operates over the LLVM-IR (instead of operating in operator-overloading or source-rewriting modes) and can thus synthesize gradients for multiple languages as long as they have

---

Function Signature	Pybind11		Julia's `ccall`		Speedup
`int fn0()`	132	±14.9	2.34	±1.24	56×
`int fn1(int)`	217	±20.9	2.35	±1.33	92×
`double fn2(int, double)`	232	±11.7	2.32	±0.189	100×
`char* fn3(int, double, char*)`	267	±28.9	6.27	±0.396	42×

Table 2. Round-trip times for calling C functions from Python (using Pybind11) and Julia (using `ccall`). All times are in nanoseconds. Round-trip times in Python include the time to resolve the function symbol, convert Python types to native C-types, invoke the function call, and return the result (including the conversion of the returned C-type to native Python types). Because C-types are binary-compatible with Julia data types, the Julia benchmark does not require type conversions. The benchmark results were collected by using an Intel Core i7-1185G7 CPU running at 3.00 GHz with Julia version 1.7.1, Python version 3.8.10, and Pybind11 version 2.9.1. All scripts required to reproduce these results are available in the reproducibility repository (Churavy et al. 2022).

an LLVM front end. This means it supports C, C++, Julia, and Rust with experimental support for Fortran. Enzyme can be used for differentiating large C++ projects as well as CUDA and HIP GPU kernels (Moses et al. 2021). Support for additional forms of parallelism (e.g., OpenMP, MPI) is part of the roadmap.

By leveraging Enzyme, users can perform cross-language automatic-differentiation and thus integrate newly developed capabilities in Julia with previously existing HPC libraries.

7.2 Calling Julia from C

Fully featured Julia HPC code can be compiled into C libraries and called from regular C applications, as shown in a proof of concept with a MultiGPU 2D heat diffusion solver written in Julia and using CUDA, MPI, and graphics called from C.*

The proof of concept shows that variables can be passed from C to Julia in a straightforward and portable manner. The example passes a GPU array allocated and initialized in the C code and an MPI communicator created in the C code to the solver written in Julia. Furthermore, support of CUDA-aware MPI that leverages RDMA, which is frequently requested in HPC, was successfully demonstrated.

Straightforward scientific visualization is possible thanks to Julia's graphics packages. The proof of concept demonstrated this by producing an animated GIF using the `Plots.jl` package from within the generated C library. For additional productivity in scientific HPC code development, Julia code that is compiled to a C library (e.g., the heat diffusion solver in the proof of concept) can also be executed within the Julia run time in an interactive manner.

The library building is enabled by the `PackageCompiler.jl` julia package (Carlsson and contributors 2022).

8 Now is the time for Julia in HPC

We are seeing a rapid uptake of Julia in technical computing. Consequently, the interest in scaling up Julia applications for HPC and designing HPC applications in Julia from the start are also on the rise.

As with every new tool in HPC, the initial adoption must overcome challenges and to some extent adapt to the unique HPC environments. It is therefore encouraging that many HPC centers are already providing Julia to their users.

8.1 For application developers

The Julia language has reached a level of maturity and stability suitable for production code. Julia's language design features native performance tools, LLVM-based just-in-time compilation, and support for parallelism and hardware accelerators, and this support makes it convenient for developing high-performance applications. Furthermore, Julia adopted many tools that enhance developer productivity, including tools for package management, code introspection, a powerful REPL, and a module system. This makes Julia one of the few high-productivity high-performance programming languages.

Historically, the adoption of programming languages in HPC has been driven by the popularity of software frameworks that are programmed in those languages. Therefore, as Julia-based frameworks rise in popularity, so will the Julia language. However, it is not necessary to wait for Julia's *killer app* because HPC frameworks also have a long history of multilanguage development (e.g., calling Fortran functions from C, calling C functions from Python). Therefore, we encourage developers to begin incorporating Julia components within existing HPC frameworks with the added value of portable access to different hardware accelerator targets.

8.2 For Julia language developers

The Julia language is uniquely suited for high-productivity, high-performance code development because it already addresses many issues of developing HPC applications in other high-productivity languages. Therefore, the work for language developers is not insurmountable. At present, the adoption challenges described in this work mainly stem from HPC hardware being similar but still different from consumer-grade hardware. For example, many HPC file systems are not optimized for loading small files, thereby resulting in slower application startup times that contribute significantly to a job's overall wall time. Also, the software

---

*https://github.com/omlins/libdiffusion

and networking environments are very different at HPC centers. Vendors often address this issue by requiring the code to be compiled with their compilers to ensure the use of system drivers—something that usually does not work out of the box and can require configuration.

The Julia community is already providing many solutions in this area and truly shines with a variety of successful and documented HPC use cases—including how deployment challenges were overcome. Julia language developers should therefore curate these use cases, incorporate solutions into the language standard (e.g., ahead-of-time compilation for demanding codes, global site configurations), and add useful examples to the Julia documentation. Finally, because the Julia language has reached a high level of maturity, the language developers should now begin to emphasize language stability.

8.3 For HPC center operators

One major adoption challenge we have encountered so far is the lack of vendor support in HPC. This was felt most acutely during the initial deployment of the OLCF's Summit supercomputer because Julia lacked support for IBM's PowerPC architecture. This is less of an issue now with architectures such as ARM's AArch64 being used in consumer devices, which provides more access and opportunity for the Julia open-source community to develop support for these architectures early on (Giordano et al. 2022). HPC centers have a history of pioneering new architectures (e.g., RISC-V) and new accelerator designs, and it is important to collaborate with vendors to garner Julia support. This will obviously benefit Julia, but because Julia is based around the open-source LLVM project, it will also lead to a better open compiler ecosystem for HPC.

China's Sunway architecture is an interesting data point. Shang et al. (2022) describes a variational quantum eigensolver written in Julia scaling up to 20 million cores. While details are sparse, we can determine that they ported Julia to the Sunway SW26010P architecture. Each SW26010P core is split into a management processing element (MPE) and 64 compute processing elements (CPEs). They developed support for running on both the MPE and CPE cores. The CPE cores are targeted in an offloading style by using the infrastructure built for Julia's general accelerator support.

Conclusion

As described here, our view is that the Julia programming language provides an excellent investment opportunity for the HPC community. Julia's value proposition prioritizes the needs of HPC in the current era: programming models that closely align with science to make HPC accessible; a coordinated ecosystem approach for packaging, testing, code instrumentation, and interactive computing; a growing community; a modern and pragmatic workflow composition strategy that interoperates with LLVM and existing HPC frameworks for simulation performance; and a powerful data science and AI unified ecosystem. Not since Fortran has a programming language been designed specifically to target the needs of the broader scientific community. Julia incorporates modern software requirements into the language to enrich the end-to-end co-design process and lower the cost of the software development cycle— from idea to performance portability. This is a pivotal time for the HPC community as it continues to march toward a more heterogeneous computing landscape in the post-Moore era, in which data-driven AI workflows become relevant for scientific discovery at scale. We believe that investing in the Julia language and enriching its ecosystem capabilities will pay dividends in easing current and future challenges associated with the increasing cost and complexity of multidisciplinary HPC endeavors.

Reproducibility

The benchmarks shown in Table 2 were run on NVIDIA P100 GPUs on the Swiss National Supercomputing Centre's Piz Daint Cray XC50 and are available in our reproducibility repository (Churavy et al. 2022). The BLAS benchmarks shown in Figure 7 were run on a single Intel Xeon Gold Skylake 6148 CPU in Noctua 1 at PC2 and are also available in our reproducibility repository (Churavy et al. 2022).

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y and Zheng X (2015) TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Allen E, Chase D, Hallett J, Luchangco V, Maessen JW, Ryu S, Steele Jr GL, Tobin-Hochstadt S, Dias J, Eastlund C et al. (2005) The Fortress language specification 139(140).

Almasi G (2011) *PGAS (Partitioned Global Address Space) Languages.* Boston, MA: Springer US. ISBN 978-0-387-09766-4, pp. 1539–1545. DOI:10.1007/978-0-387-09766-4_210. URL https://doi.org/10.1007/978-0-387-09766-4_210.

AMD (2008) ROCm HIP: Heterogeneous-Computing Interface for Portability. https://github.com/ROCm-Developer-Tools/HIP.

Antypas KB, Bard DJ, Blaschke JP, Shane Canon R, Enders B, Shankar MA, Somnath S, Stansberry D, Uram TD and Wilkinson SR (2021) Enabling discovery data science through cross-facility workflows. In: *2021 IEEE International Conference on Big Data (Big Data).* pp. 3671–3680. DOI:10.1109/BigData52589.2021.9671421.

Aroeira GJ, Davis MM, Turney JM and Schaefer III HF (2022) Fermi.jl: A Modern Design for Quantum Chemistry. *Journal of Chemical Theory and Computation* DOI:10.1021/acs.jctc.1c00719.

Backus J (1980) Programming in America in the 1950s—Some Personal Impressions. In: *A History of Computing in the twentieth century.* Elsevier, pp. 125–135.

Backus JW and Heising WP (1964) Fortran. *IEEE Transactions on Electronic Computers* EC-13(4): 382–385. DOI:10.1109/PGEC.1964.263818.

Badia S and Verdugo F (2020) Gridap: An extensible finite element toolbox in Julia. *Journal of Open Source Software* 5(52): 2520. DOI:10.21105/joss.02520.

Bard D, Snavely C, Gerhardt L, Lee J, Totzke B, Antypas K, Arndt W, Blaschke J, Byna S, Cheema R, Cholia S, Day M, Enders B, Gaur A, Greiner A, Groves T, Kiran M, Koziol Q, Lehman T, Rowland K, Samuel C, Selvarajan A, Sim A, Skinner D, Stephey L, Thomas R and Torok G (2022) The lbnl superfacility project report.

Beckingsale DA, Burmark J, Hornung R, Jones H, Killian W, Kunen AJ, Pearce O, Robinson P, Ryujin BS and Scogland TR (2019) RAJA: Portable Performance for Large-Scale Scientific Applications. In: *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC).* pp. 71–81. DOI:10.1109/P3HPC49587.2019.00012.

Ben-Nun T, Gamblin T, Hollman DS, Krishnan H and Newburn CJ (2020) Workflows are the New Applications: Challenges in Performance, Portability, and Productivity. In: *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC).* pp. 57–69. DOI:10.1109/P3HPC51967.2020.00011.

Bercea GT, McRae ATT, Ham DA, Mitchell L, Rathgeber F, Nardi L, Luporini F and Kelly PHJ (2016) A structure-exploiting numbering algorithm for finite elements on extruded meshes, and its performance evaluation in Firedrake. *Geoscientific Model Development* 9(10): 3803–3815. DOI:10.5194/gmd-9-3803-2016.

Besard T, Foket C and De Sutter B (2018) Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30(4): 827–841.

Besard T, Foket C and De Sutter B (2019) Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30(4): 827–841. DOI:10.1109/TPDS.2018.2872064.

Besard T and other contributors (2020) oneAPI.jl: Julia support for the oneAPI programming toolkit. https://github.com/JuliaGPU/oneAPI.jl.

Bezanson J, Chen J, Chung B, Karpinski S, Shah VB, Vitek J and Zoubritzky L (2018) Julia: Dynamism and performance reconciled by design. *Proceedings of the ACM on Programming Languages* 2(OOPSLA): 1–23. DOI:10.1145/3276490.

Bezanson J, Edelman A, Karpinski S and Shah VB (2017) Julia: A fresh approach to numerical computing. *SIAM Review* 59(1): 65–98. DOI:10.1137/141000671.

Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S and Zhang Q (2018) JAX: composable transformations of Python+NumPy programs. URL http://github.com/google/jax.

Breiding P and Timme S (2018) HomotopyContinuation.jl: A package for homotopy continuation in Julia. In: *International Congress on Mathematical Software.* Springer, pp. 458–465. DOI:10.1007/978-3-319-96418-8_54.

Buck I (2007) GPU computing with Nvidia CUDA. In: *ACM SIGGRAPH 2007 courses.* pp. 6–es.

Byna S, Chaarawi M, Koziol Q, Mainzer J and Willmore F (2017) Tuning hdf5 subfiling performance on parallel file systems URL https://www.osti.gov/biblio/1398484.

Byrne S, Wilcox LC and Churavy V (2021) MPI.jl: Julia bindings for the Message Passing Interface. In: *Proceedings of the JuliaCon Conferences*, volume 1. p. 68. DOI:10.21105/jcon.00068. https://github.com/JuliaParallel/MPI.jl.

Carlsson K and contributors (2022) PackageCompiler.jl: Compile your Julia package. https://github.com/JuliaLang/PackageCompiler.j.

Carter Edwards H, Trott CR and Sunderland D (2014) Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* 74(12): 3202–3216. DOI:https://doi.org/10.1016/j.jpdc.2014.07.003. URL https://www.sciencedirect.com/science/article/pii/S0743731514001257. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

Chamberlain B, Callahan D and Zima H (2007) Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21(3): 291–312. DOI:10.1177/1094342007078442. URL https://doi.org/10.1177/1094342007078442.

Christ S, Schwabeneder D, Rackauckas C, Borregaard MK and Breloff T (2022) Plots.jl – a user extendable plotting api for the julia programming language. DOI:10.48550/ARXIV.2204.08775. URL https://arxiv.org/abs/2204.08775.

Churavy V, Godoy WF, Bauer C, Ranocha H, Schlottke-Lakemper M, Räss L, Blaschke J, Giordano M, Schnetter E, Omlin S, Vetter JS and Edelman A (2022) Reproducibility repository for Bridging HPC communities through the Julia programming language. https://github.com/JuliaParallel/paper-2022-HPC. DOI:10.5281/zenodo.7236016.

Contributors JP (2015) DistributedArrays.jl: Distributed Arrays in Julia. https://github.com/JuliaParallel/DistributedArrays.jl.

Cusumano-Towner MF, Saad FA, Lew AK and Mansinghka VK (2019) Gen: A general-purpose probabilistic programming system with programmable inference. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019. New York, NY, USA: Association for Computing Machinery. ISBN 9781450367127, p. 221–236. DOI:10.1145/3314221.3314642. URL https://doi.org/10.1145/3314221.3314642.

Dagum L and Menon R (1998) OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5(1): 46–55.

Danisch S and Krumbiegel J (2021) Makie.jl: Flexible high-performance data visualization for julia. *Journal of Open Source Software* 6(65): 3349. DOI:10.21105/joss.03349. URL https://doi.org/10.21105/joss.03349.

de Graaf C and contributors (2022) PkgTemplates.jl: Create new Julia packages, the easy way. https://github.com/invenia/PkgTemplates.jl.

Dongarra J, Beckman P, Moore T, Aerts P, Aloisio G, Andre JC, Barkai D, Berthou JY, Boku T, Braunschweig B, Cappello F, Chapman B, Chi X, Choudhary A, Dosanjh S, Dunning T, Fiore S, Geist A, Gropp B, Harrison R, Hereld M, Heroux M, Hoisie A, Hotta K, Jin Z, Ishikawa Y, Johnson F, Kale S, Kenway R, Keyes D, Kramer B, Labarta J, Lichnewsky A, Lippert T, Lucas B, Maccabe B, Matsuoka S, Messina P, Michielse P, Mohr B, Mueller MS, Nagel WE, Nakashima H, Papka ME, Reed D, Sato M, Seidel E, Shalf J, Skinner D, Snir M, Sterling T, Stevens R, Streitz F, Sugar B, Sumimoto S, Tang W, Taylor J, Thakur R, Trefethen A, Valero M, van der Steen A, Vetter J, Williams P, Wisniewski R and Yelick K (2011) The International Exascale Software Project roadmap. *The International Journal of High Performance Computing Applications* 25(1): 3–60. DOI:10.1177/1094342010391989. URL https://doi.org/10.1177/1094342010391989.

Dongarra J, Graybill R, Harrod W, Lucas R, Lusk E, Luszczek P, McMahon J, Snavely A, Vetter J, Yelick K et al. (2008) DARPA's HPCS program: History, models, tools, languages. In: *Advances in Computers*, volume 72. Elsevier, pp. 1–100.

Dormand JR and Prince PJ (1980) A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics* 6(1): 19–26. DOI:10.1016/0771-050X(80)90013-3.

Dunning I, Huchette J and Lubin M (2017) JuMP: A modeling language for mathematical optimization. *SIAM review* 59(2): 295–320. DOI:10.1137/15M1020575.

Edelman A (2019) IEEE-CS sidney fernbach memorial award. SC '19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.

El-Ghazawi T, Carlson W, Sterling T and Yelick K (2005) *UPC: distributed shared memory programming*, volume 40. John Wiley & Sons.

Elrod C (2021) Roadmap to Julia BLAS and linear algebra. https://www.youtube.com/watch?v=KQ8nvlURX4M. Talk presented at JuliaCon.

Elrod C, Aluthge D, Protter M and contributors (2022) Octavian.jl: Multi-threaded BLAS-like library that provides pure Julia matrix multiplication. https://github.com/JuliaLinearAlgebra/Octavian.jl.

Elrod C and Lilly E (2019) LoopVectorization.jl: Macro(s) for vectorizing loops. https://github.com/JuliaSIMD/LoopVectorization.jl.

Enos J, Steffen C, Fullop J, Showerman M, Shi G, Esler K, Kindratenko V, Stone JE and Phillips JC (2010) Quantifying the impact of GPUs on performance and energy efficiency in HPC clusters. In: *International Conference on Green Computing.* pp. 317–324. DOI: 10.1109/GREENCOMP.2010.5598297.

Fieker C, Hart W, Hofmann T and Johansson F (2017) Nemo/Hecke: computer algebra and number theory packages for the Julia programming language. In: *Proceedings of the 2017 acm on international symposium on symbolic and algebraic computation.* pp. 157–164. DOI:10.1145/3087604.3087611.

Frost JM (2017) Calculating polaron mobility in halide perovskites. *Physical Review B* 96(19): 195202. DOI: 10.1103/PhysRevB.96.195202.

Ge H, Xu K and Ghahramani Z (2018) Turing: a language for flexible probabilistic inference. In: *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain.* pp. 1682–1690. URL http://proceedings.mlr.press/v84/ge18b.html.

Giannakou A, Blaschke JP, Bard D and Ramakrishnan L (2021) Experiences with cross-facility real-time light source data analysis workflows. In: *2021 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC).* pp. 45–53. DOI:10.1109/UrgentHPC54802.2021.00011.

Giordano M (2016) Uncertainty propagation with functionally correlated quantities.

Giordano M, Klöwer M and Churavy V (2022) Productivity meets Performance: Julia on A64FX. In: *2022 IEEE International Conference on Cluster Computing (CLUSTER).* pp. 549–555. DOI:10.1109/CLUSTER51413.2022.00072.

Godoy WF, Podhorszki N, Wang R, Atkins C, Eisenhauer G, Gu J, Davis P, Choi J, Germaschewski K, Huck K, Huebl A, Kim M, Kress J, Kurc T, Liu Q, Logan J, Mehta K, Ostrouchov G, Parashar M, Poeschel F, Pugmire D, Suchyta E, Takahashi K, Thompson N, Tsutsumi S, Wan L, Wolf M, Wu K and Klasky S (2020) Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX* 12: 100561. DOI:https://doi.org/10.1016/j.softx.2020.100561.

Gropp W, Gropp WD, Lusk E, Skjellum A and Lusk ADFEE (1999) *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press.

Hanson EP and Giordano M (2021) Code, docs, and tests: what's in the General registry? URL https://julialang.org/blog/2021/08/general-survey/.

Herbst MF, Levitt A and Cancès E (2021) DFTK: A Julian approach for simulating electrons in solids. *Proc. JuliaCon Conf.* 3: 69. DOI:10.21105/jcon.00069.

Heroux MA (2019) The Extreme-Scale Scientific Software Stack (e4s). Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Heroux MA, Carter J, Thakur R, Vetter J, McInnes LC, Ahrens J and Neely JR (2018) ECP Software Technology Capability Assessment Report DOI:10.2172/1463232. URL https://www.osti.gov/biblio/1463232.

HPCWire (2017) Julia Joins Petaflop Club. URL https://www.hpcwire.com/off-the-wire/julia-joins-petaflop-club/.

Innes M (2018) Flux: Elegant machine learning with julia. *Journal of Open Source Software* DOI:10.21105/joss.00602.

Innes M, Saba E, Fischer K, Gandhi D, Rudilosso MC, Joy NM, Karmali T, Pal A and Shah V (2018) Fashionable modelling with flux. *CoRR* abs/1811.01457. URL https://arxiv.org/abs/1811.01457.

Iravanian S, Martensen CJ, Cheli A, Gowda S, Jain A, Julia Computing, Ma Y and Rackauckas C (2022) Symbolic-numeric integration of univariate expressions based on sparse regression.

Jakob W, Rhinelander J and Moldovan D (2017) pybind11 – seamless operability between c++11 and python. Https://github.com/pybind/pybind11.

Janssens B (2022) CxxWrap.jl: Package to make C++ libraries available in Julia. https://github.com/JuliaInterop/CxxWrap.jl.

Johnson SG and contributors (2022) PyCall.jl: Package to call Python functions from the Julia language. https://github.com/JuliaPy/PyCall.jl.

Jupyter Development Team (2022) *Jupyter: Free software, open standards, and web services for interactive computing across all programming languages.* URL https://jupyter.org/.

Karpinski S (2019) The Unreasonable Effectiveness of Multiple Dispatch. https://youtu.be/kc9HwsxE1OY. Talk presented at JuliaCon.

Karrasch D, Haegeman J and contributors (2022) LinearMaps.jl: A Julia package for defining and working with linear maps, also known as linear transformations or linear operators acting on vectors. The only requirement for a LinearMap is that it can act on a vector (by multiplication) efficiently. https://github.com/JuliaLinearAlgebra/LinearMaps.jl.

Kedward LJ, Aradi B, Čertík O, Curcic M, Ehlert S, Engel P, Goswami R, Hirsch M, Lozada-Blanco A, Magnin V, Markus A, Pagone E, Pribec I, Richardson B, Snyder H, Urban J and Vandenplas J (2022) The State of Fortran. *Computing in Science & Engineering* 24(2): 63–72. DOI: 10.1109/MCSE.2022.3159862.

Kernighan BW and Ritchie DM (1988) *The C programming language.* Pearson Educación.

Ketcheson DI and Ranocha H (2021) Computing with B-series.

Kindratenko VV, Enos JJ, Shi G, Showerman MT, Arnold GW, Stone JE, Phillips JC and Hwu Wm (2009) GPU clusters for high-performance computing. In: *2009 IEEE International Conference on Cluster Computing and Workshops.* pp. 1–8. DOI:10.1109/CLUSTR.2009.5289128.

Klöwer M, Düben P and Palmer T (2020) Number formats, error mitigation, and scope for 16-bit arithmetics in weather and climate modeling analyzed with a shallow water model. *Journal of Advances in Modeling Earth Systems* 12(10): e2020MS002246. DOI:10.1029/2020MS002246.

Krämer S, Plankensteiner D, Ostermann L and Ritsch H (2018) QuantumOptics.jl: A Julia framework for simulating open quantum systems. *Computer Physics Communications* 227: 109–116. DOI:10.1016/j.cpc.2018.02.004.

Lai R and contributors (2022) RCall.jl: Call R from Julia. https://github.com/JuliaInterop/RCall.jl.

Lattner C and Adve V (2004) LLVM: a compilation framework for lifelong program analysis amp; transformation. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* pp. 75–86. DOI:10.1109/CGO.2004.1281665.

Lavrijsen WT and Dutta A (2016) High-performance python-c++ bindings with pypy and cling. In: *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC).* pp. 27–35. DOI:10.1109/PyHPC.2016.008.

Legat B, Dowson O, Garcia JD and Lubin M (2020) MathOptInterface: a data structure for mathematical optimization problems.

Lin WC and McIntosh-Smith S (2021) Comparing Julia to Performance Portable Parallel Programming Models for HPC. In: *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS).* pp. 94–105. DOI:10.1109/PMBS54543.2021.00016.

Logg A and Wells GN (2010) DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software (TOMS)* 37(2): 1–28. DOI:10.1145/1731022.1731030.

Ma Y, Gowda S, Anantharaman R, Laughman C, Shah V and Rackauckas C (2021) Modelingtoolkit: A composable graph transformation system for equation-based modeling.

Marques O and Barker A (2020) Training Efforts in the Exascale Computing Project. *Computing in Science & Engineering* 22(5): 103–107. DOI:10.1109/MCSE.2020.3010596.

Mohamad M and contributors (2022) MATLAB.jl: Calling MATLAB in Julia through MATLAB Engine. https://github.com/JuliaInterop/MATLAB.jl.

Moore GE (1998) Cramming more components onto integrated circuits. *Proceedings of the IEEE* 86(1): 82–85.

Moses W and Churavy V (2020) Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. *Advances in neural information processing systems* 33: 12472–12485.

Moses WS, Churavy V, Paehler L, Hückelheim J, Narayanan SHK, Schanen M and Doerfert J (2021) Reverse-mode automatic differentiation and optimization of GPU kernels via enzyme. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* New York, NY, USA: ACM. DOI:10.1145/3458817.3476165. To be published.

Munshi A (2009) The OpenCL specification. In: *2009 IEEE Hot Chips 21 Symposium (HCS).* IEEE, pp. 1–314.

Norton I, Qi Y and contributors (2022) Clang.jl: Julia interface to libclang. https://github.com/JuliaInterop/Clang.jl.

Numrich RW and Reid J (1998) Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum* 17(2): 1–31. DOI:10.1145/289918.289920. URL https://doi.org/10.1145/289918.289920.

Oak Ridge Leadership Computing Facility (????) Oak Ridge National Laboratory. URL https://www.olcf.ornl.gov.

Omlin S and Räss L (2019) ParallelStencil.jl: Package for writing high-level code for parallel high-performance stencil computations that can be deployed on both GPUs and CPUs. https://github.com/omlins/ParallelStencil.jl.

Omlin S, Räss L, Kwasniewski G, Malvoisin B and Podladchikov YY (2020) Solving Nonlinear Multi-Physics on GPU Supercomputers with Julia. https://youtu.be/vPsfZUqI4_0. Talk presented at JuliaCon.

Omlin S, Räss L and Utkin I (2019) ImplicitGlobalGrid.jl: Almost trivial distributed parallelization of stencil-based GPU and CPU applications on a regular staggered grid. https://github.com/eth-cscs/ImplicitGlobalGrid.jl.

Orban D, Siqueira AS and contributors (2020) LinearOperators.jl. https://github.com/JuliaSmoothOptimizers/LinearOperators.jl. DOI:10.5281/zenodo.2559295.

Ousterhout JK (1998) Scripting: higher level programming for the 21st century. *Computer* 31(3): 23–30. DOI:10.1109/2.660187.

Parashar M, Hariri S, Haupt T and Fox G (1994) A study of software development for high performance computing. In: Decker KM and Rehmann RM (eds.) *Programming Environments for Massively Parallel Distributed Systems.* Basel: Birkhäuser Basel. ISBN 978-3-0348-8534-8, pp. 107–116.

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L et al. (2019) Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32.

Pich O, Muiños F, Lolkema MP, Steeghs N, Gonzalez-Perez A and Lopez-Bigas N (2019) The mutational footprints of cancer therapies. *Nature genetics* 51(12): 1732–1740. DOI:10.1038/s41588-019-0525-5.

Plietzsch A, Kogler R, Auer S, Merino J, Gil-de Muro A, Liße J, Vogel C and Hellmann F (2022) PowerDynamics.jl - An experimentally validated open-source package for the dynamical analysis of power grids. *SoftwareX* 17: 100861. DOI:10.1016/j.softx.2021.100861.

Rackauckas C and Nie Q (2017) DifferentialEquations.jl – A performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software* 5(1): 15. DOI:10.5334/jors.151.

Rackauckas C and Nie Q (2019) Confederated modular differential equation APIs for accelerated algorithm development and benchmarking. *Advances in Engineering Software* 132: 1–6. DOI:10.1016/j.advengsoft.2019.03.009.

Rackauckas C, Singhvi A, Ma Y, Hatherly M, Jones S, Caine C, Saba E, TagBot J and Olver S (2020) Sciml/differentialequations. jl: v6. 15.0 .

Ramadhan A, Wagner G, Hill C, Campin JM, Churavy V, Besard T, Souza A, Edelman A, Ferrari R and Marshall J (2020) Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs. *Journal of Open Source Software* 5(53). DOI:10.21105/joss.02018.

Ranocha H, Schlottke-Lakemper M, Winters AR, Faulhaber E, Chan J and Gassner G (2022) Adaptive numerical simulations with Trixi.jl: A case study of Julia for scientific computing. *Proceedings of the JuliaCon Conferences* 1(1): 77. DOI:10.21105/jcon.00077.

Räss L, Omlin S and Podladchikov YY (2019) Resolving Spontaneous Nonlinear Multi-Physics Flow Localization in 3-D: Tackling Hardware Limit. URL https://developer.nvidia.com/gtc/2019/video/S9368. GTC Silicon Valley - 2019.

Räss L, Utkin I, Duretz T, Omlin S and Podladchikov YY (2022) Assessing the robustness and scalability of the accelerated pseudo-transient method. *Geoscientific Model Development* 15(14): 5757–5786. DOI:10.5194/gmd-15-5757-2022. URL https://gmd.copernicus.org/articles/15/5757/2022/.

Reed D, Gannon D and Dongarra J (2022) Reinventing High Performance Computing: Challenges and Opportunities. DOI:10.48550/ARXIV.2203.02544. URL https://arxiv.org/abs/2203.02544.

Regier J, Pamnany K, Fischer K, Noack A, Lam M, Revels J, Howard S, Giordano R, Schlegel D, McAuliffe J et al. (2018) Cataloging the visible universe through Bayesian inference in Julia at petascale. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, pp. 44–53. DOI:10.1016/j.jpdc.2018.12.008.

Revels J, Lubin M and Papamarkou T (2016) Forward-Mode Automatic Differentiation in Julia.

Reyes R and Lomüller V (2016) SYCL: Single-source C++ accelerator programming. In: *Parallel Computing: On the Road to Exascale*. IOS Press, pp. 673–682.

Rowley C (2022) Pythoncall.jl: Python and julia in harmony. URL https://github.com/cjdoris/PythonCall.jl.

Rutkowski K (2022) CBinding.jl: Automatic C interfacing for Julia. https://github.com/analytech-solutions/CBinding.jl.

Saba E and contributors (2022) BinaryBuilder.jl: Binary dependency builder for Julia. https://github.com/JuliaPackaging/BinaryBuilder.jl.

Samaroo J, Besard T, Churavy V, Lin D and other contributors (2013) AMDGPU.jl: AMD GPU (ROCm) programming in Julia. https://github.com/JuliaGPU/AMDGPU.jl.

Saraswat VA, Sarkar V and von Praun C (2007) X10: Concurrent Programming for Modern Architectures. In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07. New York, NY, USA: Association for Computing Machinery. ISBN 9781595936028, p. 271. DOI:10.1145/1229428.1229483. URL https://doi.org/10.1145/1229428.1229483.

Schäfer F, Tarek M, White L and Rackauckas C (2021) AbstractDifferentiation.jl: Backend-agnostic differentiable programming in Julia.

Schlottke-Lakemper M, Winters AR, Ranocha H and Gassner GJ (2021) A purely hyperbolic discontinuous Galerkin approach for self-gravitating gas dynamics. *Journal of Computational Physics* : 110467DOI:10.1016/j.jcp.2021.110467.

Schnetter E and contributors (2016) SIMD.jl: Explicit SIMD vector operations for Julia. https://github.com/eschnett/SIMD.jl.

Scholtz J and Wiedenbeck S (1990) Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human–Computer Interaction* 2(1): 51–72. DOI:10.1080/10447319009525970.

Shalf JM and Leland R (2015) Computing beyond Moore's Law. *Computer* 48(12): 14–23. DOI:10.1109/MC.2015.374.

Shang H, Shen L, Fan Y, Xu Z, Guo C, Liu J, Zhou W, Ma H, Lin R, Yang Y, Li F, Wang Z, Zhang Y and Li Z (2022) Large-Scale simulation of quantum computational chemistry on a new sunway supercomputer .

Shrestha N, Botta C, Barik T and Parnin C (2020) Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language? In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. pp. 691–701.

Stevens R, Taylor V, Nichols J, Maccabe AB, Yelick K and Brown D (2020) AI for Science: Report on the Department of Energy (DOE) Town Halls on Artificial Intelligence (AI) for Science DOI:10.2172/1604756. URL https://www.osti.gov/biblio/1604756.

Straßel D, Reusch P and Keuper J (2020) Python workflows on hpc systems. In: *2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. pp. 32–40. DOI:10.1109/PyHPC51966.2020.00009.

Stroustrup B (2013) *The C++ programming language*. Pearson Education.

The HDF Group (2000-2010) Hierarchical data format version 5. URL http://www.hdfgroup.org/HDF5.

van der Plas F, Dral M, Berg P, Παναγιωτης Γεωργακοπουλος, Bochenski N, disberd, Lungwitz B, Huijzer R, Zhang E, Schneider FSS, Weaver I, Rogerluo, Kadowaki S, Ling J, Wu Z, Burns C, Gerritsen J, Novosel R, Supanat, Moon Z, pupuis, Abbott M, Bauer N, Bouffard P, Terasaki S, Polasa S, TheCedarPrince and fghzxm (2022) fonsp/pluto.jl: v0.17.7. DOI:10.5281/zenodo.5889169. URL https://doi.org/10.5281/zenodo.5889169.

Verdugo F and Badia S (2021) The software design of Gridap: a finite element package based on the Julia JIT compiler.

Vetter JS, Brightwell R, Gokhale M, McCormick P, Ross R, Shalf J, Antypas K, Donofrio D, Humble T, Schuman C, Van Essen B, Yoo S, Aiken A, Bernholdt D, Byna S, Cameron K, Cappello F, Chapman B, Chien A, Hall M, Hartman-Baker R, Lan Z, Lang M, Leidel J, Li S, Lucas R, Mellor-Crummey J, Peltz Jr P, Peterka T, Strout M and Wilke J (2018) Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity DOI:10.2172/1473756.

Weitz JS, Beckett SJ, Coenen AR, Demory D, Dominguez-Mirazo M, Dushoff J, Leung CY, Li G, Măgălie A, Park SW et al. (2020) Modeling shield immunity to reduce COVID-19 epidemic spread. *Nature medicine* 26(6): 849–854. DOI:10.1038/s41591-020-0895-3.

Wienke S, Springer P, Terboven C et al. (2012) OpenACC—first experiences with real-world applications. In: *European Conference on Parallel Processing.* Springer, pp. 859–870.

Williams C (2016) How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript. URL https://www.theregister.com/2016/03/23/npm_left_pad_chaos/.

Zhu S, AlAwar N, Erez M and Gligoric M (2021) Dynamic generation of python bindings for hpc kernels. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* pp. 92–103. DOI:10.1109/ASE51524.2021.9678726.