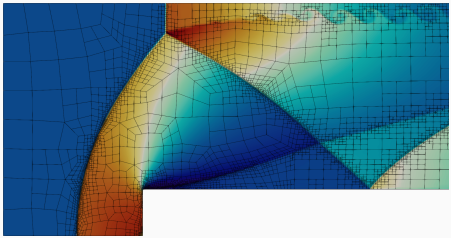
High-performance computing with Julia using Trixi.jl

An experience report

Michael Schlottke-Lakemper

Julia for HPC, 22th September 2022

High-Performance Computing Center Stuttgart (HLRS)



Where does this talk come from?

- ∼8 years of experience in "traditional" high-performance computing
  - Developed adaptive, high-order simulation code in C++
  - Hybrid parallelization with MPI and OpenMP
  - Efficient scalability up to 450,000 cores
- ∼2.5 years of experience with Julia development (since Julia v1.3)
  - Core developer of Trixi.jl
  - (Co-)created ∼10 Julia packages (small to largish)
- Main research interests:
  - Numerical simulation of multi-physics problems (CFD)
  - Research software engineering for HPC

Table of contents

Trixi.jl

Trixi.jl

- Adaptive high-order simulation framework for hyperbolic PDEs (MIT license)
- 7 developers, 20+ students, 7+2 papers
- Goals: extensibility, usability, performance
- Integration with Julia ecosphere:
  - `OrdinaryDiffEq.jl`: time integration
  - `ForwardDiff.jl`: automatic differentiation
  - `Plots.jl`, `Makie.jl`: plotting
  - `LoopVectorization.jl`: performance
  - `Polyester.jl`: multithreading

https://github.com/trixi-framework/Trixi.jl

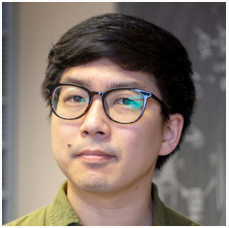
Main contributors

Michael Schlottke-Lakemper

Gregor J. Gassner

Andrew R. Winters

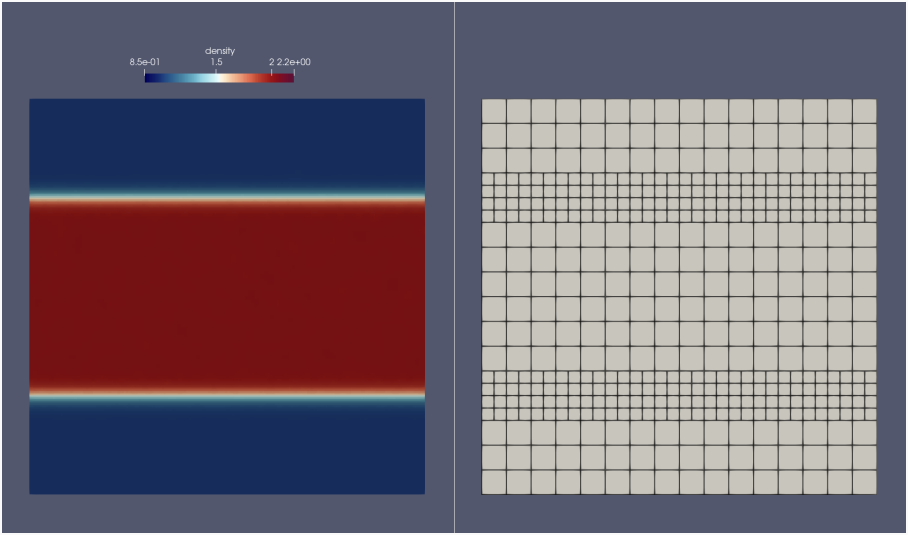
Hendrik Ranocha



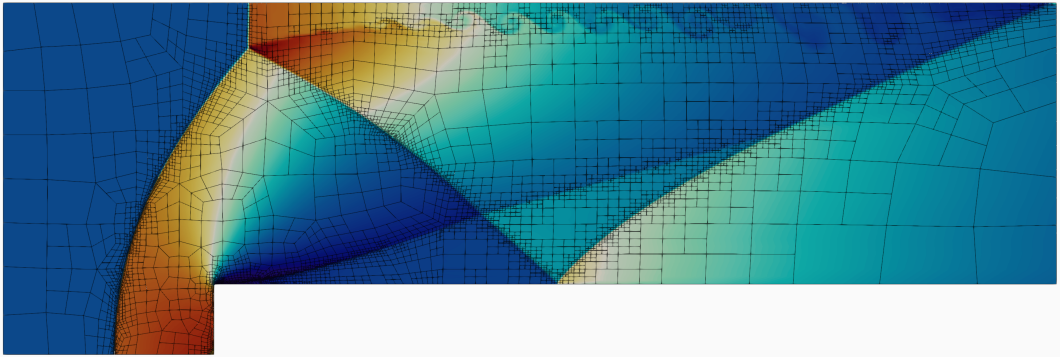
Jesse Chan

Many others:
Erik Faulhaber
Lars Christmann
Niklas Neher
...

Kelvin-Helmholtz flow instability with adaptive mesh refinement

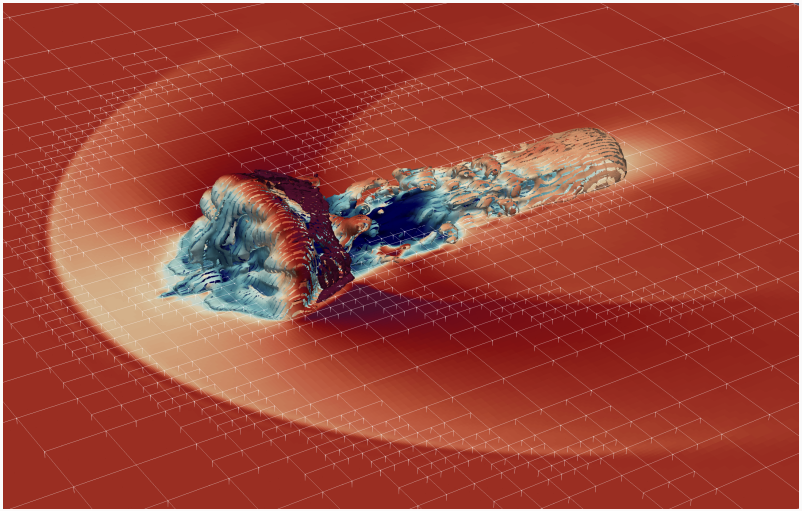


Trixi.jl simulation of front-facing step in wind tunnel at Mach 3



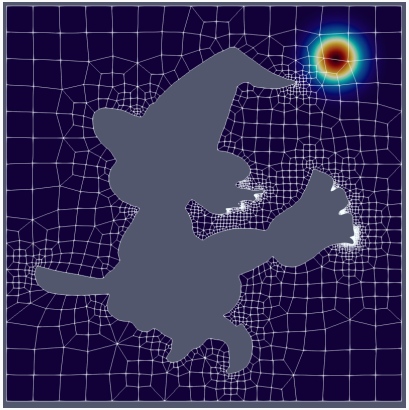
Credit: Andrew R. Winters, https://www.youtube.com/watch?v=glAug1aIxio

Cold gas cloud interacting with hot supersonic flow



Major features of Trixi.jl

- Native support for 1D/2D/3D simulations
  - Unstructured and structured meshes
  - Cartesian and curvilinear meshes
  - Adaptive mesh refinement
- High order in space and time
- Nodal DGSEM discretization
  - Kinetic energy-preserving/entropy-stable split forms
  - Entropy-stable shock capturing
  - Positivity-preserving limiting
- Multiple governing equations: compressible Euler, MHD, hyperbolic diffusion, acoustics, and more

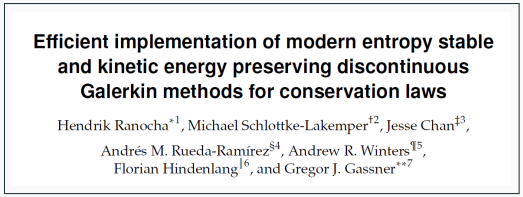


Credit: Andrew R. Winters

Serial performance and optimization

Paper: Efficient implementation of discontinuous Galerkin methods

- LGL-DGSEM with flux differencing
- Compressible Euler equations in 2D/3D
- 2 codes:
  - Trixi.jl (Julia)
  - FLUXO (Fortran)
- 3 mesh types:
  - unstructured, Cartesian
  - structured, curved
  - unstructured, curved



arXiv: 2112.10517

repro: tinyurl.com/ecperf

Evaluate impact of various performance improvement strategies

- Focus on two numerical two-point fluxes
- Flux 1: Shima et al.
  - kinetic energy preserving (KEP)
  - pressure equilibrium preserving
  - "cheap"
- Flux 2: Ranocha
  - kinetic energy preserving
  - pressure equilibrium preserving
  - entropy conserving (EC)
  - "expensive" (logarithmic mean)

Efficient implementation of modern entropy stable and kinetic energy preserving discontinuous Galerkin methods for conservation laws

Hendrik Ranocha[*1], Michael Schlottke-Lakemper[†2], Jesse Chan[‡3], Andrés M. Rueda-Ramírez[§4], Andrew R. Winters[¶5], Florian Hindenlang[|6], and Gregor J. Gassner[**7]

arXiv: 2112.10517

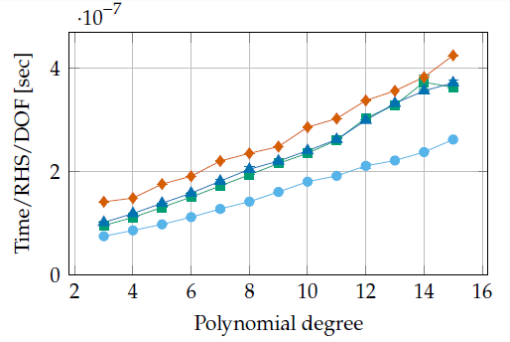
repro: tinyurl.com/ecperf

Shima, Kuya, Tamaki, Kawai, J Comput Phys, 2020. doi:10.1016/j.jcp.2020.110060

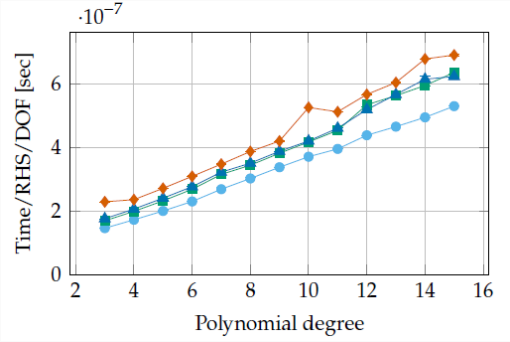
Ranocha, J Sci Comput, 2018. doi:10.1007/s10915-017-0618-1

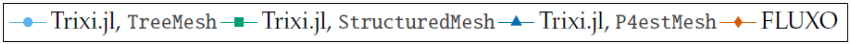
Base performance for 3D flux differencing computations

KEP flux by Shima et al.



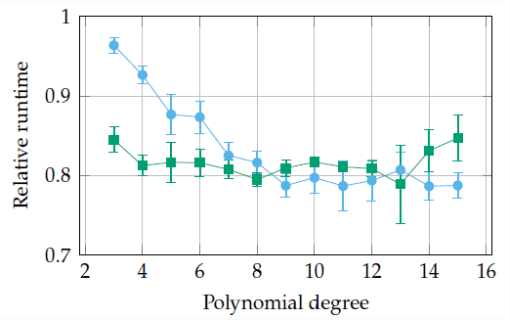
EC flux by Ranocha



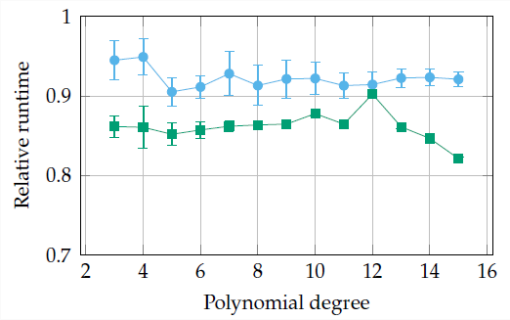


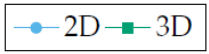
Performance improvement using precomputed primitive variables

KEP flux by Shima et al.

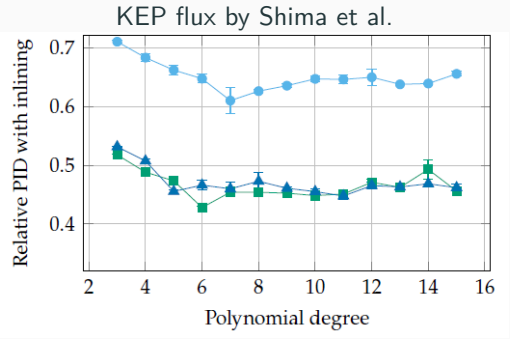


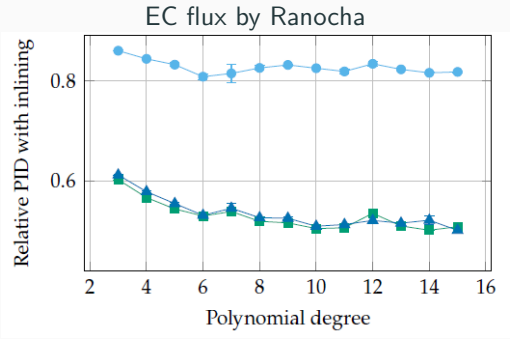
EC flux by Ranocha

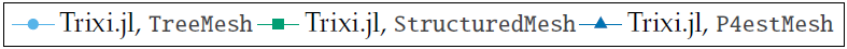




Performance of inlined vs. non-inlined flux functions

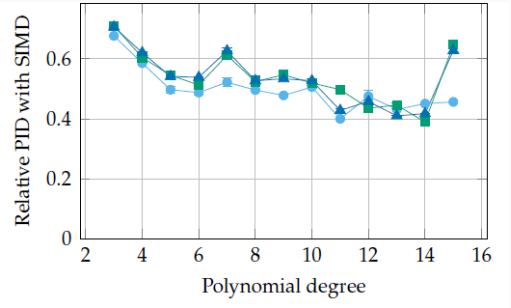




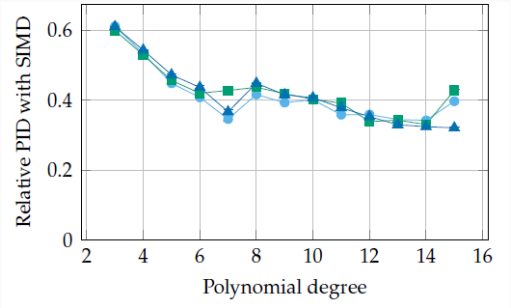


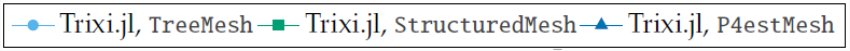
Performance of SIMD optimized code

KEP flux by Shima et al.



EC flux by Ranocha





Final PIDs at $p = 3$	KEP flux by Shima et al.	EC flux by Ranocha
Cartesian mesh	$2.6 \cdot 10^{-8}$ s	$4.2 \cdot 10^{-8}$ s
Unstructured, curved mesh	$4.3 \cdot 10^{-8}$ s	$6.5 \cdot 10^{-8}$ s

What about *serial performance*?

Lessons learned

Similar (or better) performance with Julia compared to Fortran. Algorithmic improvements still relevant.

Extensibility and binary dependencies

What about *extensibility*?

Precomputed primitive variables

- Create new data type `PrimVars` to hold primitive variables
- Specialize Trixi's internal functions on new data type, e.g., `cons2prim(...)`

```
1    cons2prim(u,           equation::CompressibleEuler3D) // default implementation
2    cons2prim(u::PrimVars, equation::CompressibleEuler3D) // new
```

What about *extensibility*?

SIMD optimization

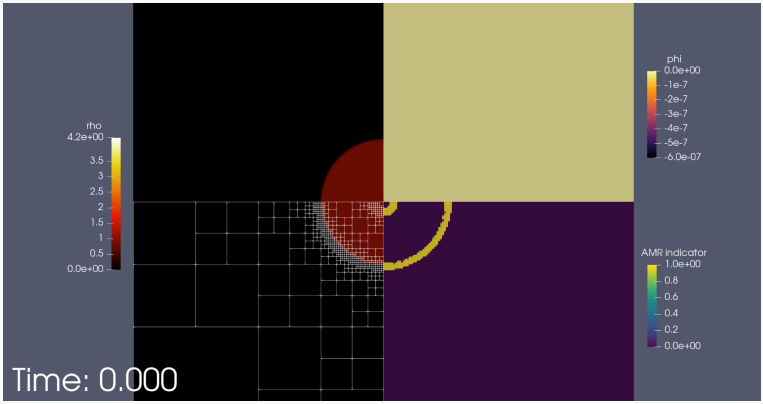
- Create new flux function `flux_ranocha_turbo`
- Specialize Trixi's internal functions on new flux type, e.g., `split_form_kernel!`
- Put data layout conversion into specialized functions

```
1    split_form_kernel!(du, ..., volume_flux::typeof(flux_ranocha_turbo))
```

Note: All modifications possible from "outside" of Trixi.jl

What about *extensibility*?

- Flow-gravity simulation with shock capturing and adaptive mesh refinement
- Multi-physics coupling with less than 350 lines of code



Schlottke-Lakemper, Winters, Ranocha, Gassner, J Comput Physics, 2021.

doi:10.1016/j.jcp.2021.110467 | arXiv:2008.10593

What about *extensibility*?

Lessons learned

Design your code layout for Julia (and not for C/C++/Fortran).

Exploit rapid prototyping capabilities for performance tuning.

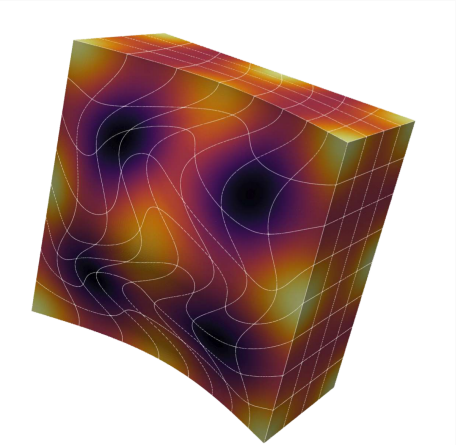
What about *binary dependencies*?

- Install pre-compiled binaries via `Pkg`
- Can be used for libraries and executables

Example: adaptive meshes with p4est

- Wrapper package P4est.jl
- Auto-installs OS-specific binaries
- Works on Linux, MacOS, Windows

`https://github.com/trixi-framework/P4est.jl`



Lessons learned

Re-use existing (parallel) code from C/C++/Fortran.

Parallel usage and performance

Does it scale?

- Julia code for computational fluid dynamics: ✓ (Trixi.jl)
- Parallelized with MPI: ✓
- Supercomputer available: ✓ (Hawk @ HLRS)

©Trixi.jl authors, Ben Derzian, HLRS

Hawk and Vulcan

- High-Performance Computing Center Stuttgart (HLRS)
- Hawk → capability computing (AMD EPYC Rome, ∼720k cores)
- Vulcan → industry computing (various Intel Xeon, 10k-20k cores)
- Julia 1.7.2, Julia 1.8.0 available as modules

https://www.hlrs.de/solutions/systems

HLRS Hawk

HLRS Vulcan

Where to put the Julia folder

- `JULIA_DEPOT_PATH` envvar specifies location for package files, precompilation files...
- Put it in home directory (NFS)?
  - Julia crashed when using >2000 MPI ranks
- Put it on parallel file system (Lustre)?
  - >10,000 small files
  - Issues: slow on `Pkg` update, quota limits

Lessons learned

Figure out depot location before doing parallel runs.

Observations for MPI and multithreading

- Generally high memory usage of Julia runtime: Trixi.jl + OrdinaryDiffEq.jl
  - 360 MiB after loading
  - 510 MiB after compilation
- Hybrid parallelization (MPI + multithreading) support may be system dependent
- Non-standard MPI implementations might require additional efforts

Lessons learned

Test which MPI implementation to use.
Verify that multithreading works as expected.

Using Julia with a system MPI may require user action

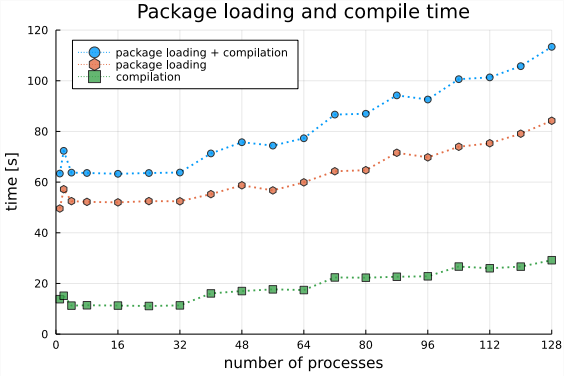
- Need to replace MPI-enabled JLL-provided binaries by system binaries (e.g., MPI.jl, HDF5.jl)
- Need to regenerate C bindings for libraries due to MPI ABI change (e.g., P4est.jl)

Lessons learned

Document setup procedures for your users and yourself.
Have a look at MPItrampoline (`https://github.com/eschnett/MPItrampoline`).

Runtime overhead due to code loading/compilation

- Code loading times increase due to file system pressure
- Also code compilation can be slower in parallel
- General rule: non-parallel operations increase computational cost

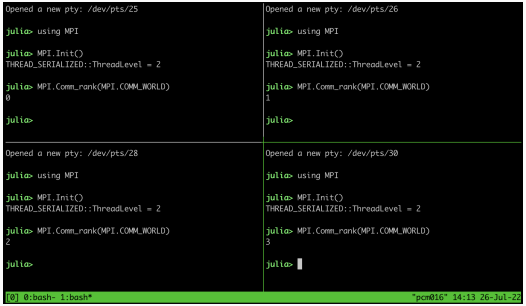


Lessons learned

Use PackageCompiler.jl to create custom Julia sysimage (`https://github.com/JuliaLang/PackageCompiler.jl`).
Try to use HPC/Spindle to avoid I/O bottleneck (`https://github.com/hpc/Spindle`).

Restrictions for interactive computing with MPI

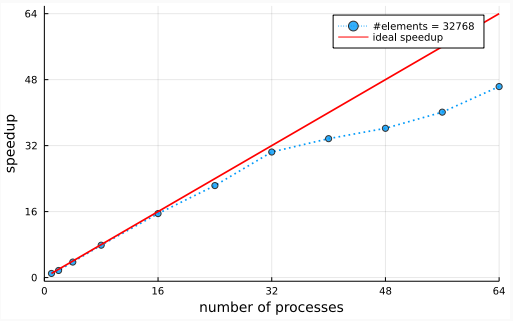
- No inherent MPI support for the REPL
- Long load/compilation times
- Parallel development is sub-optimal (debugging is even less fun)



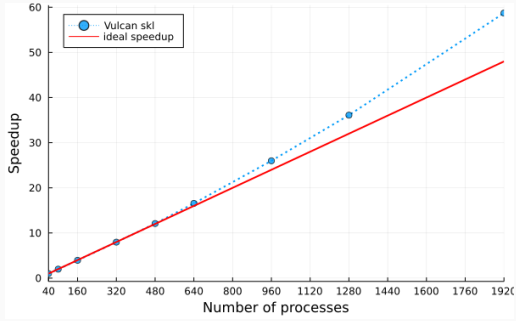
Lessons learned

Run MPI in `tmux` via `tmpi` as a workaround (`https://github.com/Azrael3000/tmpi`).
Be patient.

Strong scaling results for Trixi.jl



Intra-node scaling on Hawk (64x)

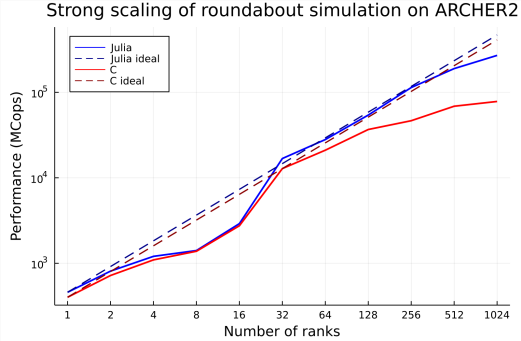


Multi-node scaling on Vulcan (40x)

Lessons learned

Parallel performance of Julia with MPI behaves similarly to C++/Fortran codes.

Strong scaling results for TrafficMPI.jl on ARCHER2



- Simple parallel simulation of roundabout traffic
- Credit: Mose Giordano, UCL
- `https://github.com/giordano/TrafficMPI.jl`

Lessons learned

Direct comparison of parallel Julia/C shows near-identical behavior.

Julia HPC community

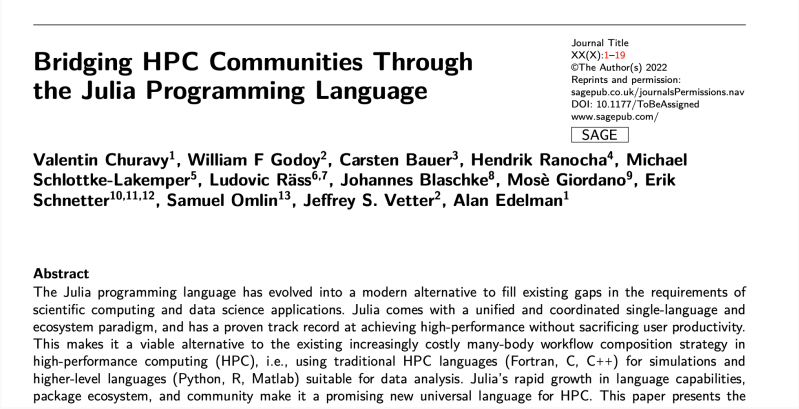
Interact with the Julia HPC community

- Reach out via the Julia Slack workspace
  - `#hpc`
  - `#distributed`
  - `#performance-helpdesk`
- Join the monthly Julia HPC call (next meeting: Thu, Sep 28, 8pm CEST)
- Get involved in key HPC packages such as MPI.jl, CUDA.jl
- Check out the "Julia for HPC" minisymposim at JuliaCon 2022

Lessons learned

You are not alone.
Reach out and you will get help.

Joint effort: paper on Julia for HPC



Bridging HPC Communities Through the Julia Programming Language

Journal Title
XX(X):1–19

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

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

Abstract
The Julia programming language has evolved into a modern alternative to fill existing gaps in the requirements of scientific computing and data science applications. Julia comes with a unified and coordinated single-language and ecosystem paradigm, and has a proven track record at achieving high-performance without sacrificing user productivity. This makes it a viable alternative to the existing increasingly costly many-body workflow composition strategy in high-performance computing (HPC), i.e., using traditional HPC languages (Fortran, C, C++) for simulations and higher-level languages (Python, R, Matlab) suitable for data analysis. Julia's rapid growth in language capabilities, package ecosystem, and community make it a promising new universal language for HPC. This paper presents the

Lessons learned

There are opportunities for collaborative work & publications.

Conclusions

- Serial and parallel performance of Julia is on par with C/C++/Fortran
- Extensibility is great for performance optimization
- Code reuse through binary interoperability with C/Fortran
- New challenges compared to traditional HPC languages
- Parallel development toolchain not as mature yet

Feel free to reach out!

m.schlottke-lakemper@hlrs.de, Julia Slack, Trixi Slack, ...