Controlling parallel simulations in Julia from C/C++/Fortran programs with libtrixi

Michael Schlottke-Lakemper[*], Benedict Geihe[†]

[*] Applied and Computational Mathematics, RWTH Aachen University
[*] High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart
[†] Numerical Simulation, University of Cologne

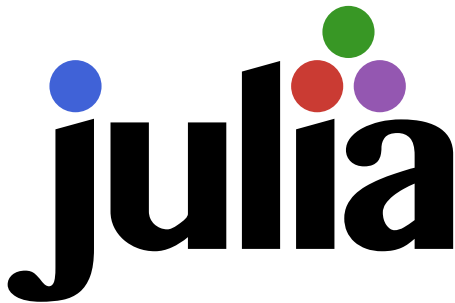
deRSE24, Würzburg, Germany, 6th March 2024

Programming languages in HPC

- What is HPC?
- Mostly C/C++/Fortran codes (>90% of programs, >99% of SLOC)
- Past decade: Python/Lua used as glue code
- Most large software packages with Python interface
- But what about the other way around?

Enter Julia

- Modern language for high-performance scientific computing
- Combine ease-of-use, interoperability, and performance
- Dynamic typing and JIT compilation
- Goal: solve two-language problem



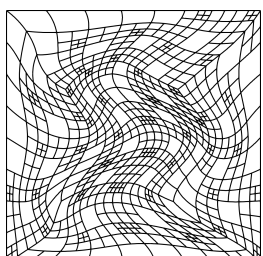
Well-established approach: Julia → C/Fortran

- Re-use of existing software straightforward
- Examples C libraries: MPI, HDF5, p4est, t8code
- Seamless installation and reproducibility through Pkg.jl, Yggdrasil

Example: adaptive meshes with p4est

- Wrapper package P4est.jl
- Auto-installs binaries with MPI support
- Works on Linux, macOS, Windows

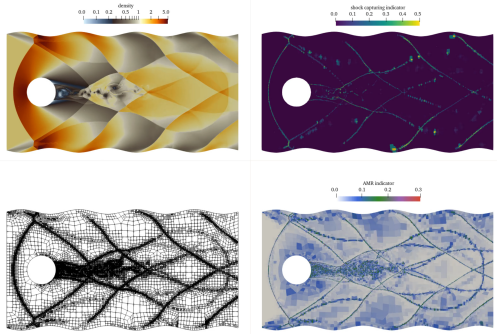
`https://github.com/trixi-framework/P4est.jl`



Trixi.jl: parallel adaptive simulations in Julia

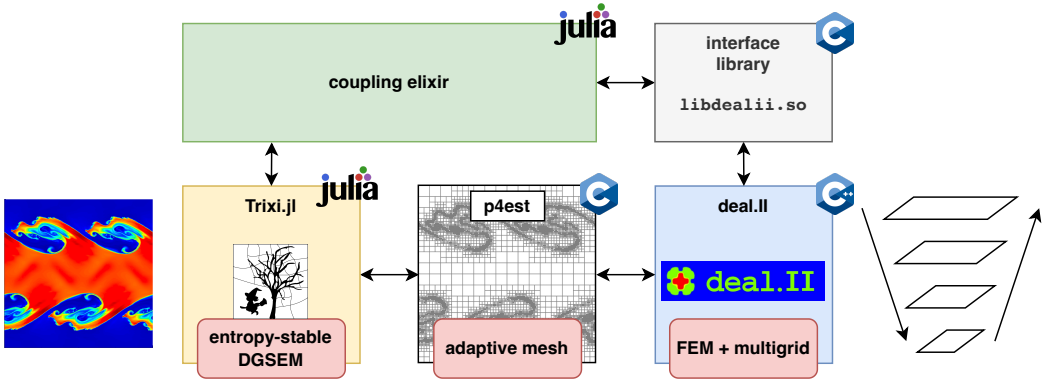
- Adaptive high-order simulation framework for conservation laws (MIT license)
- Developed in Julia programming language
- Single-command reproducibility
- ∼20 main developers  ,
  15+ papers, 20+ BSc/MSc/PhD theses
- Focus: extensibility, usability, performance

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



Example for interoperability in HPC: calling deal.II from Trixi.jl

- Couple state-of-the-art solvers with parallel mesh adaptivity (proof of concept)
- Accessibility through single-command setup (multi-language workflow)
- Works in parallel (MPI) and with adaptivity



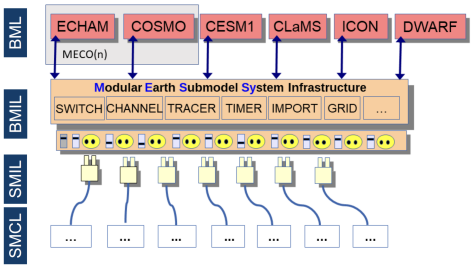
What about the other way around?

Our goal: Upgrade legacy code with new applications

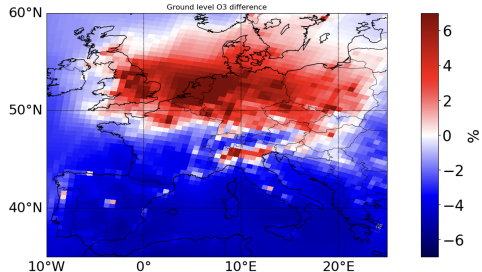
- Keep original C/C++/Fortran applications
- Extend with new features written in more accessible, user-friendly language
- Do not restrict performance or parallelizability

Interoperability in extreme-scale HPC: calling Trixi.jl from MESSy

- "Adaptive Earth system modelling with strongly reduced computation time for exascale-supercomputers" (ADAPTEX; BMBF call "SCALEXA"; 2023–2026)
- Trixi.jl as dynamical core in MESSy (Modular Earth Submodel System)



Kerkweg et al. (2010)



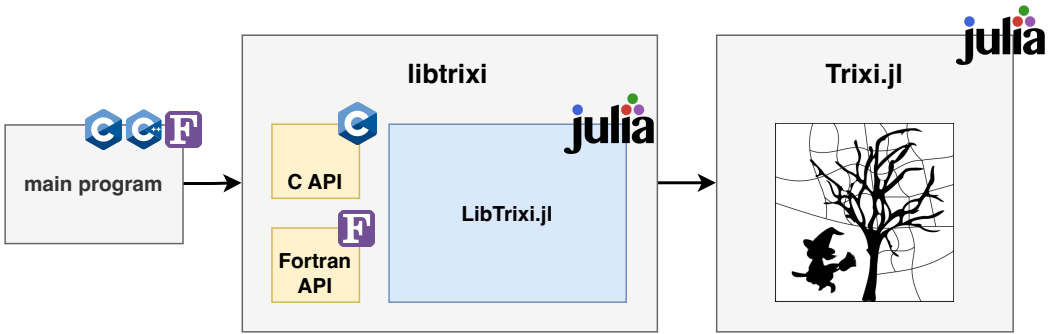
Mertens et al. (2021)

Main challenges for the project

- How to shoehorn Julia's flexibility to a static C/Fortran API?
- Does it work with MPI?
- Is there a performance impact?
- Are there other restrictions?

Overall setup

- main program (written in any language able to use C/Fortran interface)
- libtrixi: interface layer (written in C/Fortran and Julia)
- Trixi.jl: numerical simulation framework (written in Julia)



Example: get element count (part 1: main program)

- Initialization: load libelixir and store simulation state
- Handle: integer value to address multiple simulation states

```
handle = trixi_initialize_simulation("path/to/libelixir.jl")
...
trixi_nelements(handle)
```

Combine components for simulation setup in an “libelixir”

```
using Trixi, LibTrixi

function init_simstate()
  equations = CompressibleEulerEquations2D(1.4)
  solver = DGSEM(polydeg = 3, surface_flux = flux_lax_friedrichs)
  mesh = P4estMesh((8, 8), (-1.0, -1.0), (1.0, 1.0), polydeg = 3,
                   initial_refinement_level = 1)

  my_init(x, t, equations) = ...
  semi = SemidiscretizationHyperbolic(mesh, equations, my_init, solver)

  ...

  return SimulationState(semi, integrator)
end
```

Example: get element count (part 2: libtrixi, Fortran API)

- Fortran API only thin wrapper around C code (interface block)

```
1 interface
2   integer(c_int) function trixi_nelements(handle) bind(c)
3     integer(c_int), value, intent(in) :: handle
4   end function
5 end interface
```

Example: get element count (part 3: libtrixi, C API)

- C API obtains and calls function pointers from Julia
- For performance: store function pointers during initialization

```
1  int trixi_nelements(int handle) {
2    const char* command = "@cfunction(trixi_nelements, Cint, (Cint,))";
3    jl_value_t* res = jl_eval_string(command);
4
5    int (*nelements)(int) = (void *)jl_unbox_voidpointer(res);
6
7    return nelements(handle);
8  }
```

Example: get element count (part 4: LibTrixi.jl, external API)

- Julia function retrieves simulation state and calls internal API

```
1 Base.@ccallable function trixi_nelements(simstate_handle::Cint)::Cint
2     simstate = load_simstate(simstate_handle)
3     return trixi_nelements_jl(simstate)
4 end
```

Example: get element count (part 5: LibTrixi.jl, internal API)

- Internal API function unpacks data wrappers and calls Trixi.jl function (`nelements`)

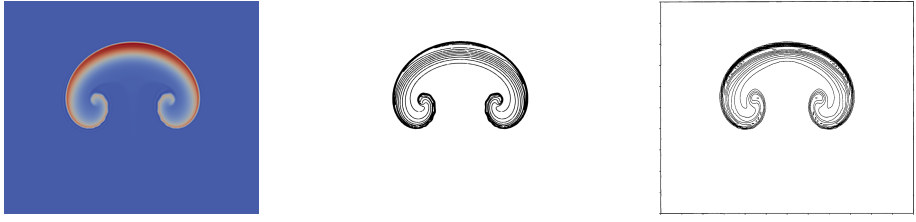
```
1 function trixi_nelements_jl(simstate)
2     _, _, solver, cache = mesh_equations_solver_cache(simstate.semi)
3     return nelements(solver, cache)
4 end
```

How to actually call Julia from C/Fortran?

- Default: use C interface of Julia runtime (shown here)
- Advanced: use PackageCompiler.jl to skip C API of libtrixi (long compilation)
- Extreme: use StaticCompiler.jl to skip Julia runtime (not possible here)

Rising thermal perturbation

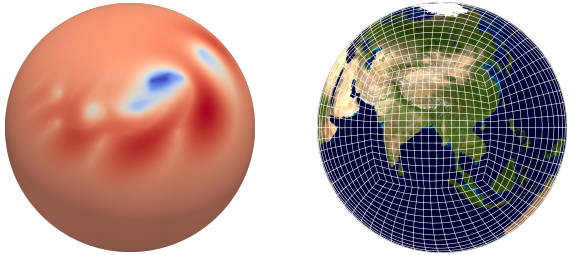
- Standard benchmark test case[1]
- 2D atmosphere in hydrostatic balance, initial circular perturbation in potential temperature
- Simulated using libtrixi from Fortran main program on 4 MPI ranks



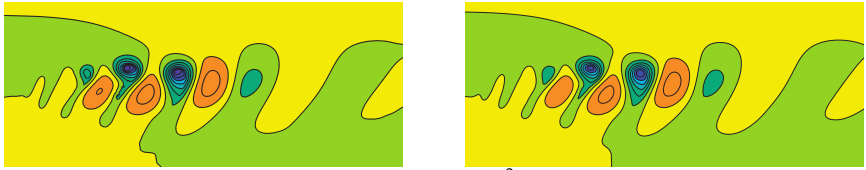
Potential temperature and contour lines with libtrixi compared to result by Bryan and Frisch.

[1] Bryan, G. H., and J. M. Fritsch, Mon. Wea. Rev., 130, 2002, doi:10.1175/1520-0493(2002)130<2917:ABSFMN>2.0.CO;2

Baroclinic instability on cubed sphere (16 MPI ranks, 2.5 mio. cells)



Ground pressure with libtrixi and cubed-sphere mesh projected on a picture of the Earth.



Surface pressure with libtrxi (left) and reference solution[2] (MCore; right). Credit: Erik Faulhaber.

[2] Ullrich, Melvin et al., QJ Roy Meteor Soc 140.682, 2014, doi:10.1002/qj.2241

Conclusions

- Controllling Julia code from C/C++/Fortran works
- No apparent limitations regarding performance or parallelism
- Advanced Julia constructs require interface layer
- Julia just another language in multi-language projects

Conclusions

- Controllling Julia code from C/C++/Fortran works
- No apparent limitations regarding performance or parallelism
- Advanced Julia constructs require interface layer
- Julia just another language in multi-language projects

Thank you for your attention!

Repro repo: `https://github.com/trixi-framework/talk-2024-deRSE-libtrixi`