libtrixi

# Serving legacy Fortran codes with simulations in Julia

*Benedict Geihe* [1], Gregor Gassner [1], Michael Schlottke-Lakemper [2]

[1] University of Cologne, Division of Mathematics

[2] Augsburg University, Institute of Mathematics

Juliacon 2024, Eindhoven, July 12th, 2024

# libtrixi

- Interface library to CFD solver `Trixi.jl`
- From C / C++ / Fortran to Julia
- Used for HPC simulations

  in earth system modeling

- Why?
- How?
- What?

# Why ?

- Project *ADAPTEX*

  *Adaptive Earth system modelling with strongly reduced computation time for exascale-supercomputers*

- Make legacy ESM codes exascale-ready
- Keep original code
- Add new features written in a high-level language
- Do not sacrifice performance
- Support heterogeneous HPC environments

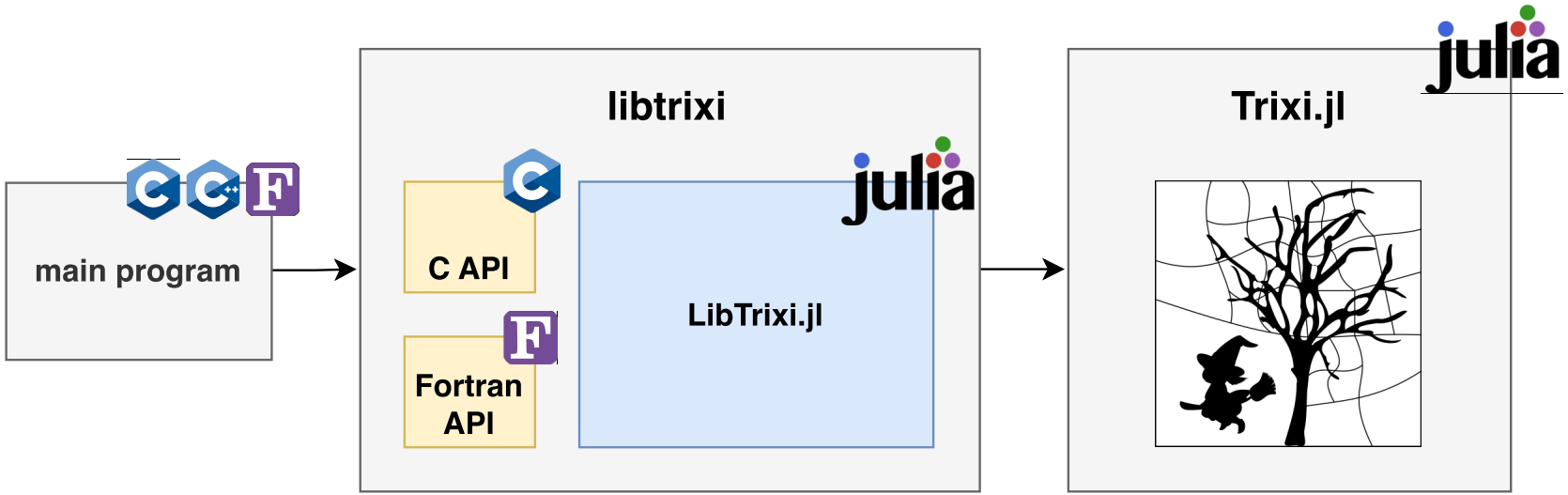
SPONSORED BY THE

Federal Ministry
of Education
and Research

**Funded by
the European Union**

NextGenerationEU

# How?


main program
libtrixi
C API
Fortran API
LibTrixi.jl
Trixi.jl

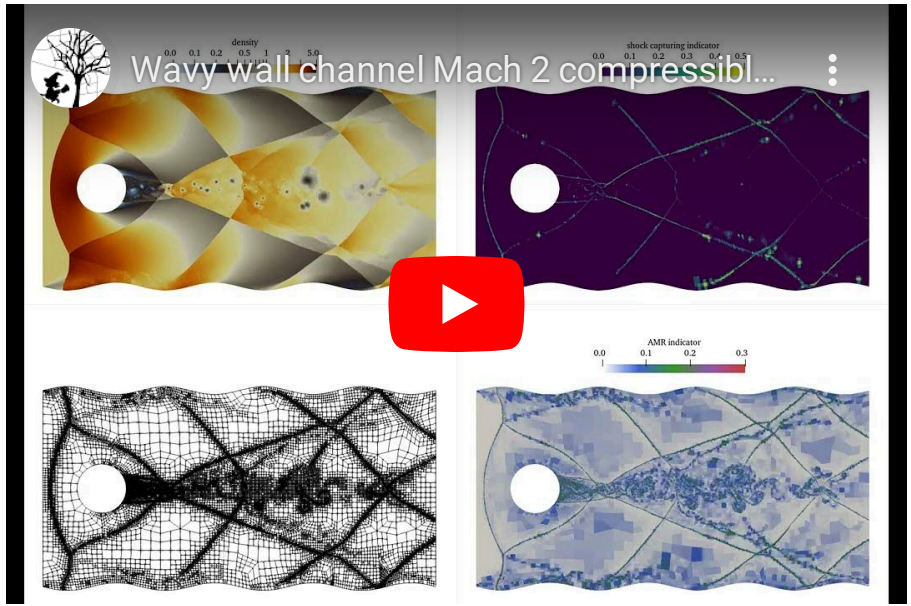
# Trixi.jl

- Simulation framework for conservation laws
- Adaptive high-order DG method
- Entropy stable and

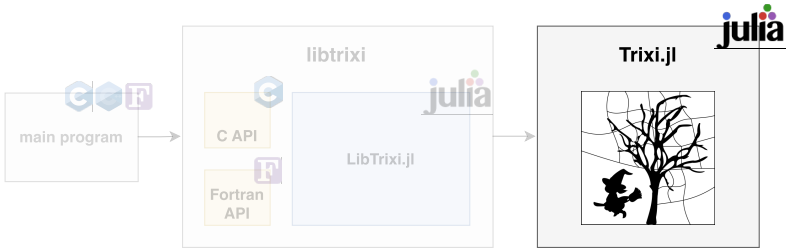
  entropy conserving schemes
- ~20 main developers,

  ~75k lines of code

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



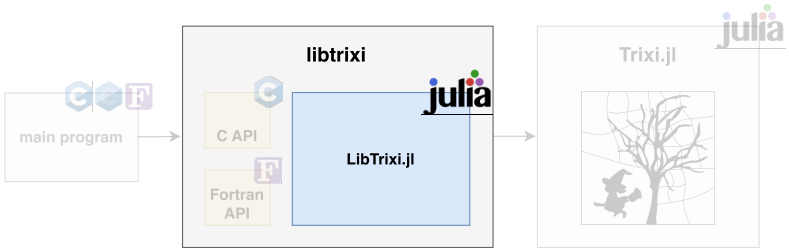
# Trixi.jl




- Configure simulations through code (*elixir*)

```
using Trixi, LibTrixi
function init_simstate()
  equations = CompressibleEulerEquations2D(gamma = 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)
  my_init(x, t, equations) = ...
  semi = SemidiscretizationHyperbolic(mesh, equations, my_init, solver)
  tspan = (0.0, 1.0)
  ode = semidiscretize(semi, tspan)
  # OrdinaryDiffEq.jl
  integrator = init(ode, ...)
  return SimulationState(semi, integrator)
end
```

# Julia API



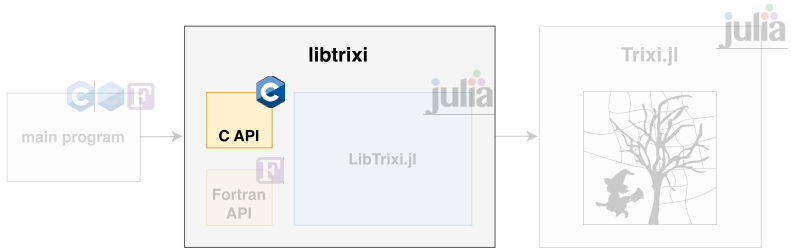

- Translate complex data
- Call `Trixi.jl` function
- Expose a C-callable function

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

Base.@ccallable function trixi_nelements(simstate_handle::Cint)::Cint
    simstate = load_simstate(simstate_handle)
    return trixi_nelements_jl(simstate)
end

trixi_nelements_cfptr() = @cfunction(trixi_nelements, Cint, (Cint,))
```

# C API



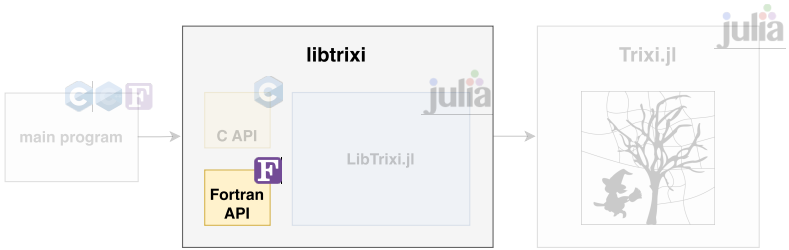

- Obtain Julia function pointers once

```
const char* command = "trixi_nelements_cfptr()";
jl_value_t* res = jl_eval_string(command);
trixi_function_pointers[TRIXI_FPTR_NELEMENTS] = (void *) jl_unbox_voidpointer(res);
```

- Get stored function pointer
- Call function

```
int trixi_nelements(int handle) {
    int (*nelements)(int) = trixi_function_pointers[TRIXI_FPTR_NELEMENTS];
    return nelements(handle);
}
```

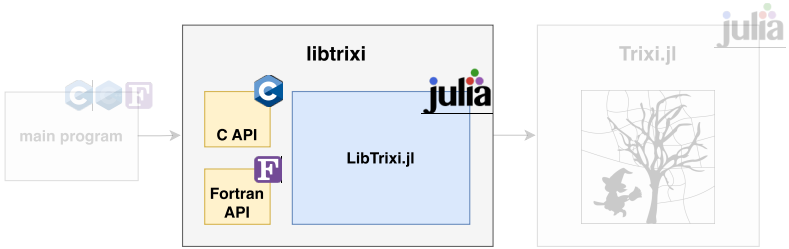
# Fortran API




- Fortran wrapper around C code

```
interface
  integer(c_int) function trixi_nelements(handle) bind(c)
    integer(c_int), value, intent(in) :: handle
  end function
end interface
```

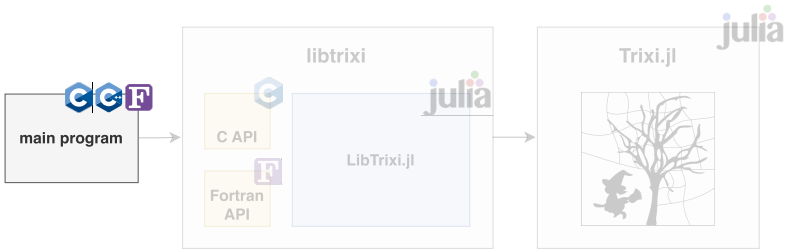
# GenLibTrixi.jl




- *Cause it's easy once you know how it's done*

```
functions:
- name: nelements
    doc_brief:  Return number of local elements
    args:
      - name:   handle
        ctype:  int
        intent: in
        doc:    simulation handle
    return:
      ctype: int
```

# Usage




- Build `libtrixi`, set up Julia project
- Include module, compile, link `libtrixi.so`

```
program trixi_controller_simple_f
  use LibTrixi
  ! Initialize
  call trixi_initialize(julia_runtime_dir)
  handle = trixi_initialize_simulation(elixir)
  do
    if ( trixi_is_finished(handle) ) exit
    call trixi_step(handle)
  end do
  ! Finalize
  call trixi_finalize_simulation(handle)
  call trixi_finalize()
end program
```

# Performance

- Benchmark problems in 2D and 3D
- Compressible Euler equations

| | `Trixi.jl` | `LibTrixi.jl` | `libtrixi` |
|---|---|---|---|
| 2D | 7.9 s | 11.3 s | 11.2 s |
| 3D | 327.0 s | 332.0 s | 329.0 s |

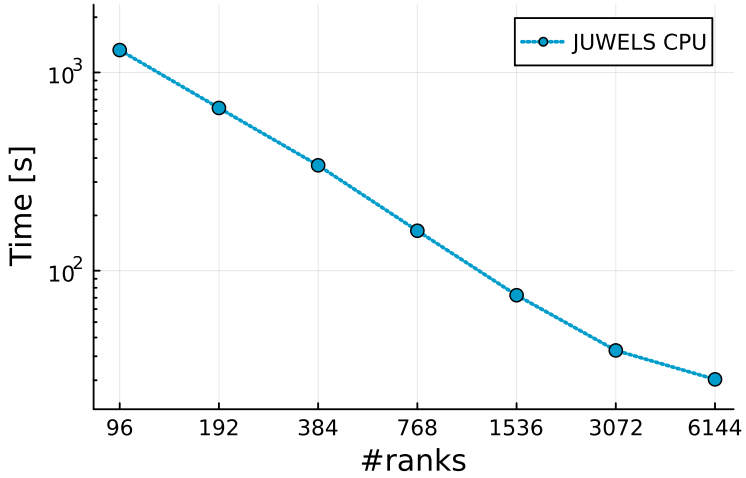
- Negligible overhead for practical applications
- Performance of `Trixi.jl` on par with classical codes

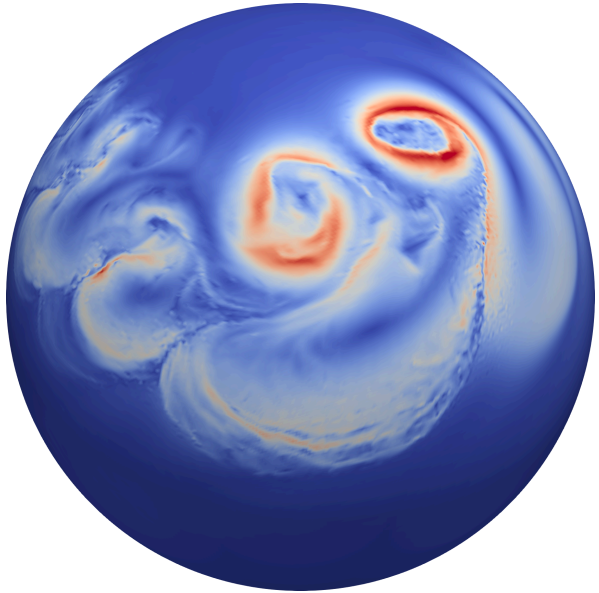
H. Ranocha et al. "Efficient Implementation of Modern Entropy Stable and Kinetic Energy Preserving Discontinuous Galerkin Methods for Conservation Laws". ACM Trans. Math. Softw. (2023).

# Results

- Use from `MESSy` (*Modular Earth Submodel System*), written in Fortran
- Baroclinic instability, 98 304 cells, 21 233 664 DOFs
- Leverage distributed compute capabilities

  (based on `MPI.jl`)

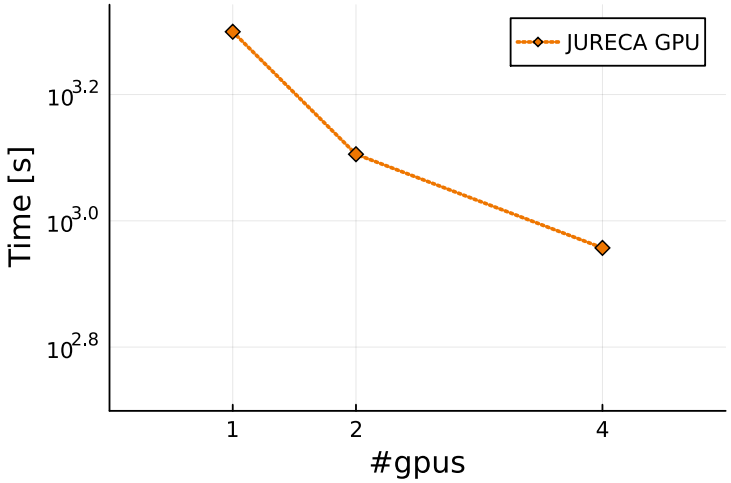


# Results

- Leverage GPU offloading capability

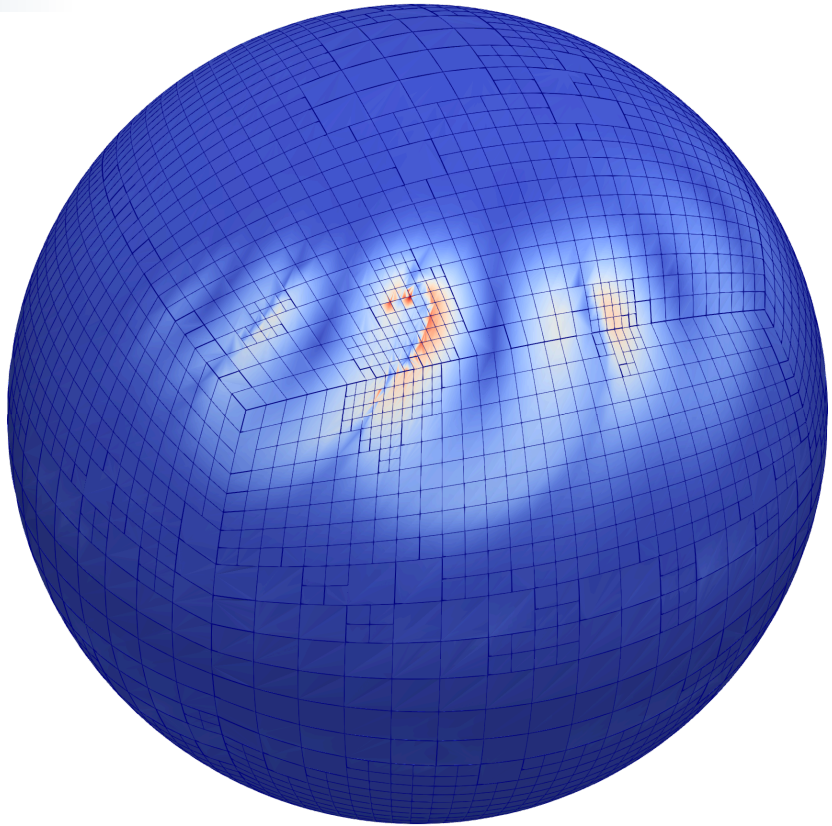
  (based on `KernelAbstractions.jl`, `Adapt.jl`)




```
ode = semidiscretize(semi, tspan)
```

# What is next?

- Leverage adaptive mesh refinement
- Improve multi-GPU implementation
- Simulation in single precision
- Advanced models for atmospheric flows



# Thanks!

- Julia
- Trixi
- Michael Schlottke-Lakemper
- Lars Christmann
- Jülich Supercomputing Centre

Check out our reproducibility repository!

`github.com/trixi-framework/talk-2024-juliacon-libtrixi`