Paderborn
Center for
Parallel
Computing

Bridging Domain Science and HPC with Julia

Carsten Bauer @ HPC Café, NHR@FAU

May 7, 2024

We don't always speak the same language

Domain Science

High-Performance Computing

Language Barrier

Julia aims to solve the "two-language problem"

Domain Science

High-Performance Computing

Gradual transition

"Julia: come for the syntax, stay for the speed"

nature

What I'll talk about

Strengths and *weaknesses* of Julia,
to give you a basis for deciding whether Julia could be of interest to you.

1. Julia's Strengths
2. Julia's Weaknesses
3. The Julia HPC Community

Julia’s Strengths

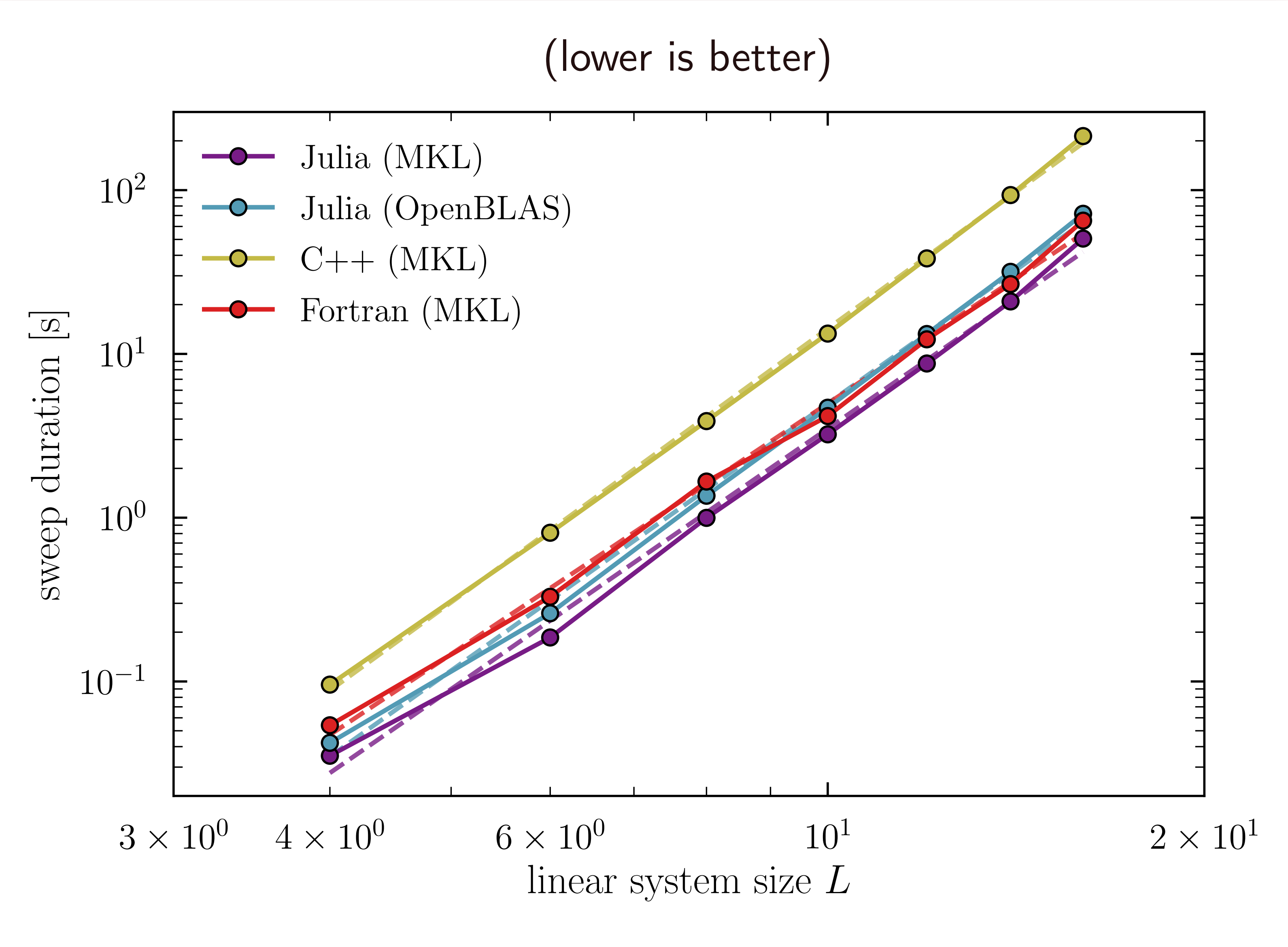
Julia code can be fast and scalable.

Type inference

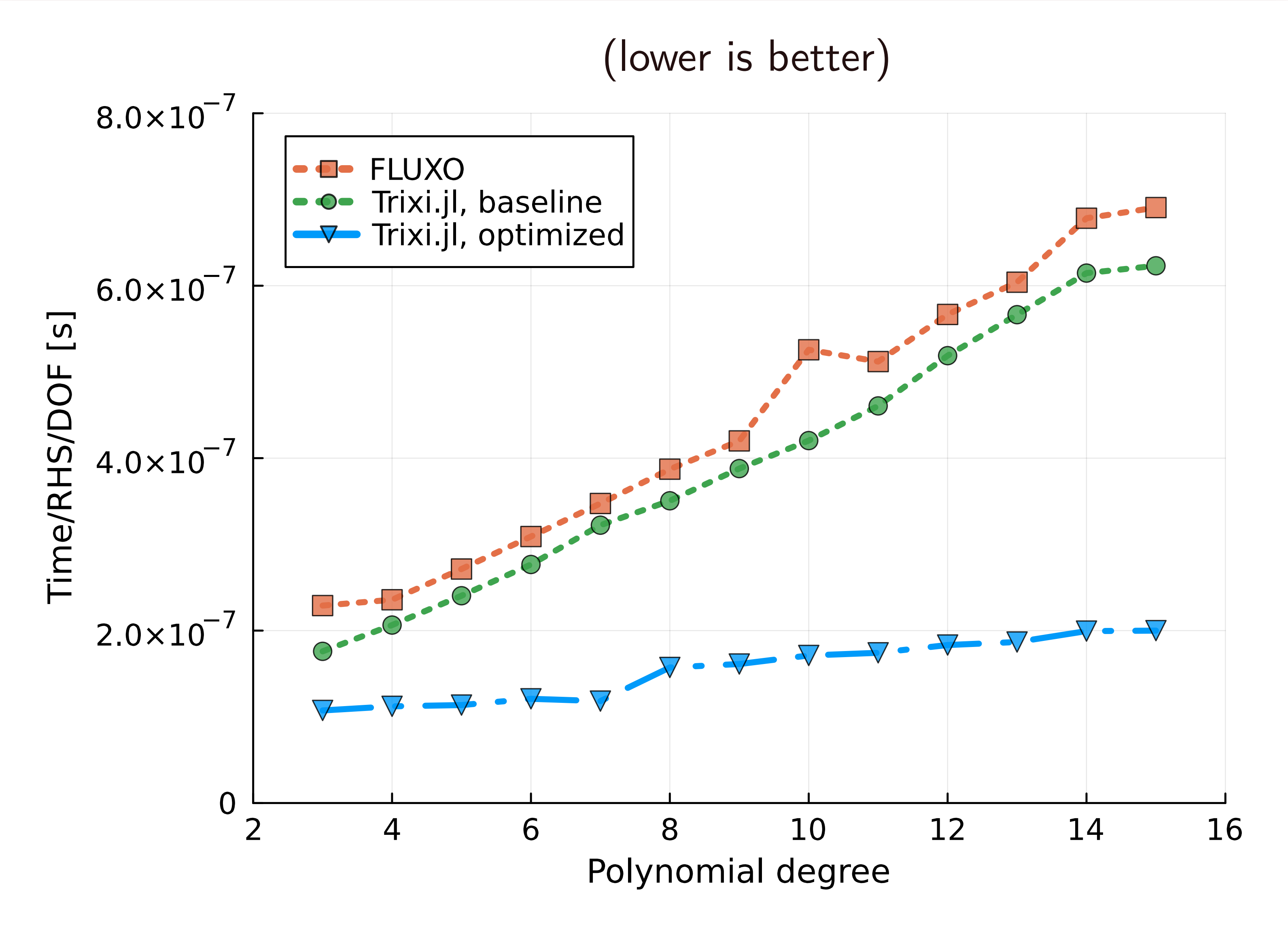
Compilation via LLVM

MPI support

Julia code can match the performance of C/Fortran

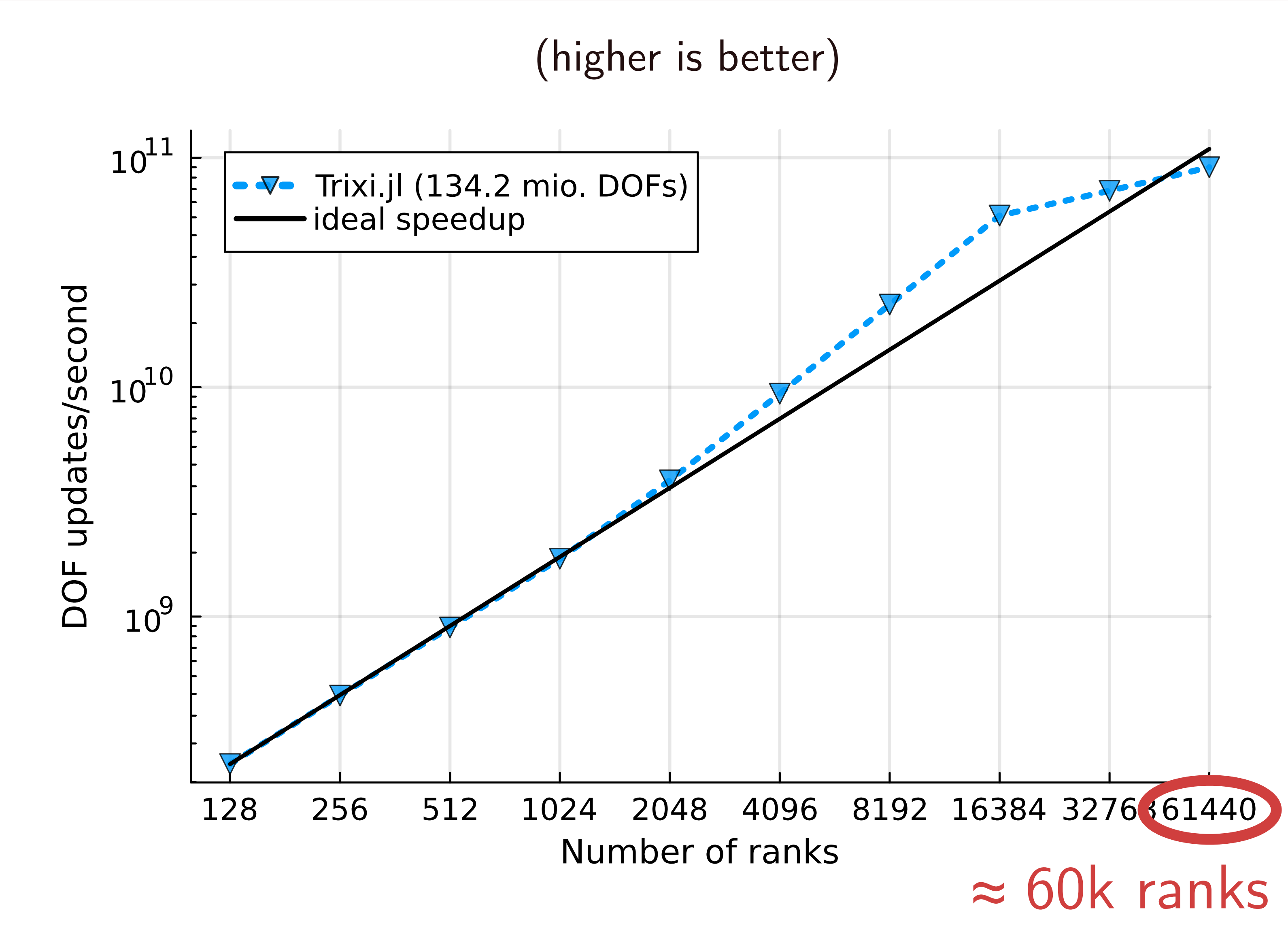


MonteCarlo.jl (DQMC)

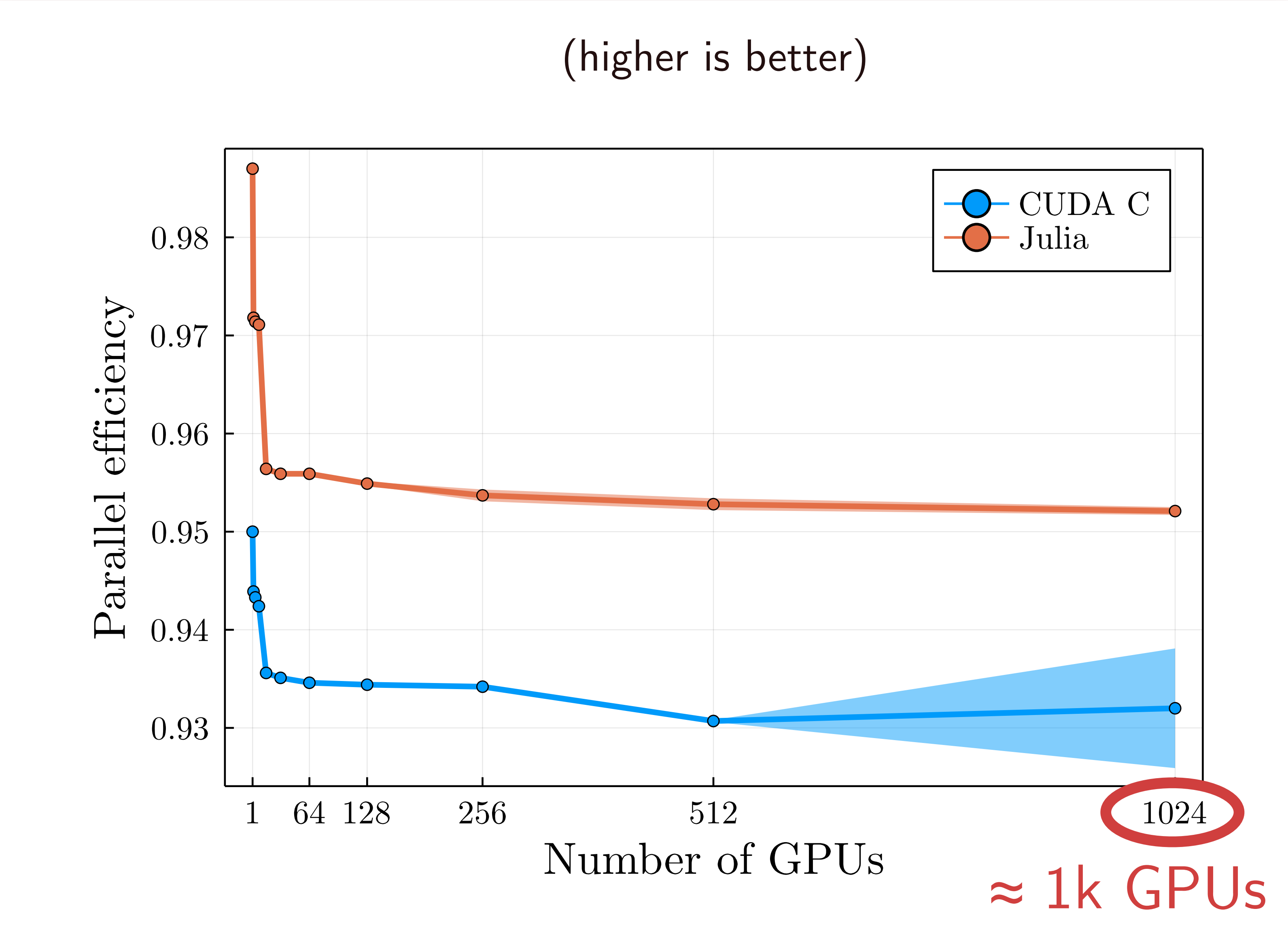


Trixi.jl (CFD)

Good scaling of PDE codes



Trixi.jl (Multi-CPU)



ParallelStencil.jl (Multi-GPU)

Julia is interactive and convenient.

Powerful REPL, Jupyter, ...

Great math support

Best-in-class package manager

LIKWID can be used interactively in a notebook

Counting Flops

```
x = rand(10_000);
```

```
function computation(x)
    x .+ x
end;
```

Counted Flops: 10000

How?

```
using LIKWID
```

```
metrics, events = @perfmon "FLOPS_DP" computation(x);
```

Compute from derived metrics

```
10000
```

```
begin
    flops_per_second = metrics["FLOPS_DP"][1]["DP [MFLOP/s]"] * 1e6
    runtime = metrics["FLOPS_DP"][1]["Runtime (RDTSC) [s]"]
    flops = round(Int, flops_per_second * runtime)
end
```

Paderborn Center for Parallel Computing
NHR FAU
Erlangen National High Performance Computing Center

Threads can be *pinned* interactively

Pin the Julia threads →

Visualize →

1st CPU

2nd CPU

```
julia> pinthreads(:sockets)

julia> threadinfo()

System: 128 cores (2-way SMT), 2 sockets, 8 NUMA domains

| 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
  16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
  32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
  48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
  128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,
  144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,
  160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,
  176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191 |
| 64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,
  80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,
  96,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,
  112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,
  192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,
  208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,
  224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,
  240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255 |

# = Julia thread, # = HT, # = Julia thread on HT, | = Socket seperator

Julia threads: 64
├ Occupied CPU-threads: 64
└ Mapping (Thread => CPUID): 1 => 0, 2 => 64, 3 => 1, 4 => 65, 5 => 2, ...

julia>
```

Threads can be *pinned* interactively

Pin the Julia threads

Visualize

NUMA domains

```
julia> pinthreads(:numa)

julia> threadinfo(; groupby=:numa)

System: 128 cores (2-way SMT), 2 sockets, 8 NUMA domains

| 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
  128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143 |
| 16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
  144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159 |
| 32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
  160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175 |
| 48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,
  176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191 |
| 64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,
  192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207 |
| 80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,
  208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223 |
| 96,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,
  224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239 |
| 112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,
  240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255 |

# = Julia thread, # = HT, # = Julia thread on HT, | = NUMA seperator

Julia threads: 64
├ Occupied CPU-threads: 64
└ Mapping (Thread => CPUID): 1 => 0, 2 => 16, 3 => 32, 4 => 48, 5 => 64, ...

julia>
```

Offers great package management and portability

Laptop

```
→  ~/myproject  tree
.
├── Manifest.toml
├── Project.toml
└── code.jl

0 directories, 3 files

→  ~/myproject  cat Project.toml
[deps]
CUDA = "052768ef-5323-5732-b1bb-66c8b64840ba"
DifferentialEquations = "0c46a032-eb83-5123-abaf-570d42b7fbaa"
MKL = "33e6dc65-8f57-5167-99aa-e5a354878fb2"
MPI = "da04e1cc-30fd-572f-bb4f-1f8673147195"

→  ~/myproject
```

HPC Cluster

```
→  bauerc@n2login3 myproject  julia --project
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.7.2 (2022-02-06)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

(myproject) pkg> st
      Status `~/myproject/Project.toml`
→ [052768ef] CUDA v3.11.0
→ [0c46a032] DifferentialEquations v7.1.0
→ [33e6dc65] MKL v0.5.0
→ [da04e1cc] MPI v0.19.2
        Info packages marked with → not downloaded, use `instantiat
e` to download

(myproject) pkg> instantiate
```

(Using **system software** is supported.)

Array abstractions for easy GPU computing

CPU

```
julia> using BenchmarkTools

julia> function axpy!(y, a, x)
           y .= a .* x .+ y
       end
axpy! (generic function with 1 method)

julia> a = rand(Float32);

julia> x = rand(Float32, 2^22);

julia> y = rand(Float32, 2^22);

julia> @btime axpy!(y, a, x);
  1.700 ms (0 allocations: 0 bytes)
```

GPU

```
julia> using BenchmarkTools, CUDA

julia> function axpy!(y, a, x)
           y .= a .* x .+ y
       end
axpy! (generic function with 1 method)

julia> a = rand(Float32);

julia> x = CUDA.rand(Float32, 2^22);

julia> y = CUDA.rand(Float32, 2^22);

julia> @btime CUDA.@sync axpy!(y, a, x);
  44.254 μs (54 allocations: 1.33 KiB)
```

($\lesssim$ 10% slower than CUBLAS)

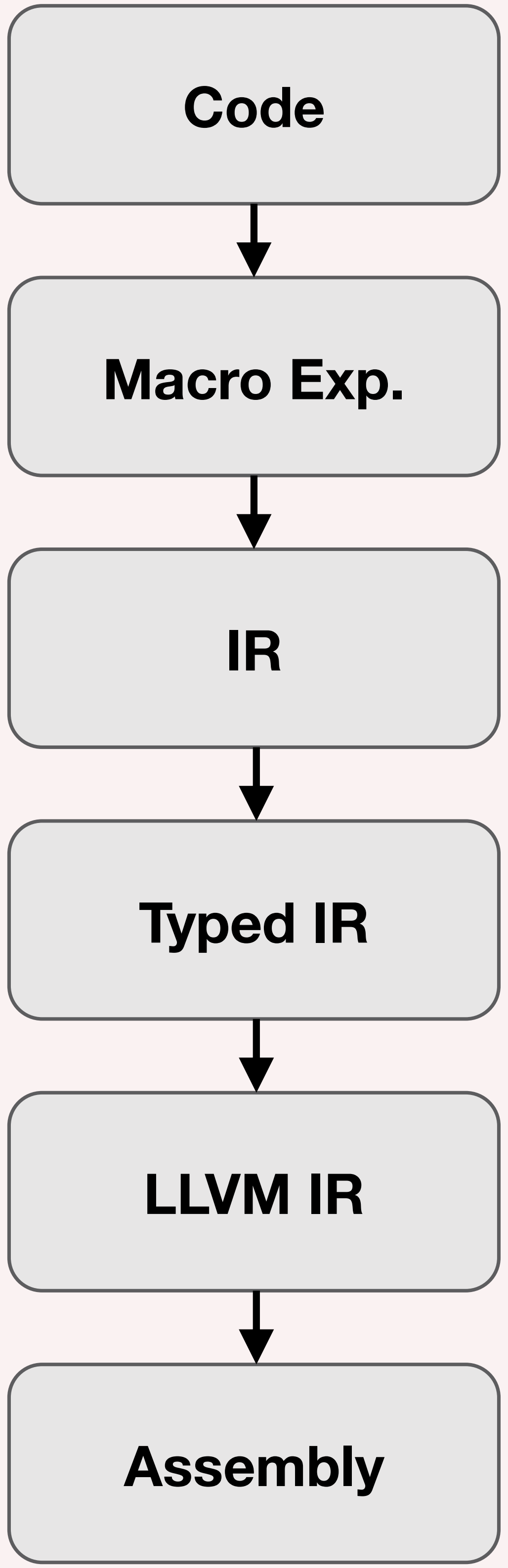
Julia invites you to gradually delve deeper.

Entirely open source

Julia is (mostly) written in Julia

Great introspection tools

Insight into different code levels




```
@which
@less
@edit

@macroexpand

@code_lowered

@code_warntype
@code_typed

@code_llvm

@code_native
```

Insight into different code levels

Source code →

```julia
julia> function gauss_sum()
           x = 0
           for i in 1:100
               x += i
           end
           return x
       end
gauss_sum (generic function with 1 method)
```

LLVM code →

```
julia> @code_llvm debuginfo=:none gauss_sum()
define i64 @julia_gauss_sum_3887() #0 {
top:
  ret i64 5050
}
```

Native code →

```
julia> @code_native debuginfo=:none dump_module=false gauss_sum()
        .text
        push    rbp
        mov     rbp, rsp
        mov     rax, qword ptr [r13 + 16]
        mov     rax, qword ptr [rax + 16]
        mov     rax, qword ptr [rax]
        mov     eax, 5050
        pop     rbp
        ret
        nop     word ptr cs:[rax + rax]
```

Julia's Weaknesses

HPC with Julia is currently a niche.

Limited support by vendors and HPC centers

Few people maintain many core packages

Still maturing

Achieving high performance can be tricky.

Garbage collection

Type instabilities

Task-based multithreading

Avoid type instabilities in performance critical code

```
julia> function type_stable()
           x = 5
           y = sqrt(x)
           return y
       end
type_stable (generic function with 1 method)

julia> @code_warntype type_stable()
MethodInstance for type_stable()
  from type_stable() @ Main REPL[18]:1
Arguments
  #self#::Core.Const(type_stable)
Locals
  y::Float64
  x::Int64
Body::Float64
1 ─      (x = 5)
│        (y = Main.sqrt(x::Core.Const(5)))
└──      return y::Core.Const(2.23606797749979)
```

Random type!

```
julia> function type_instable()
           x = rand([5, 1.2, "3.0"])
           y = sqrt(x)
           return y
       end
type_instable (generic function with 1 method)

julia> @code_warntype type_instable()
MethodInstance for type_instable()
  from type_instable() @ Main REPL[16]:1
Arguments
  #self#::Core.Const(type_instable)
Locals
  y::Any
  x::Any
Body::Any
1 ─ %1 = Base.vect(5, 1.2, "3.0")::Vector{Any}
│        (x = Main.rand(%1))
│        (y = Main.sqrt(x))
└──      return y
```

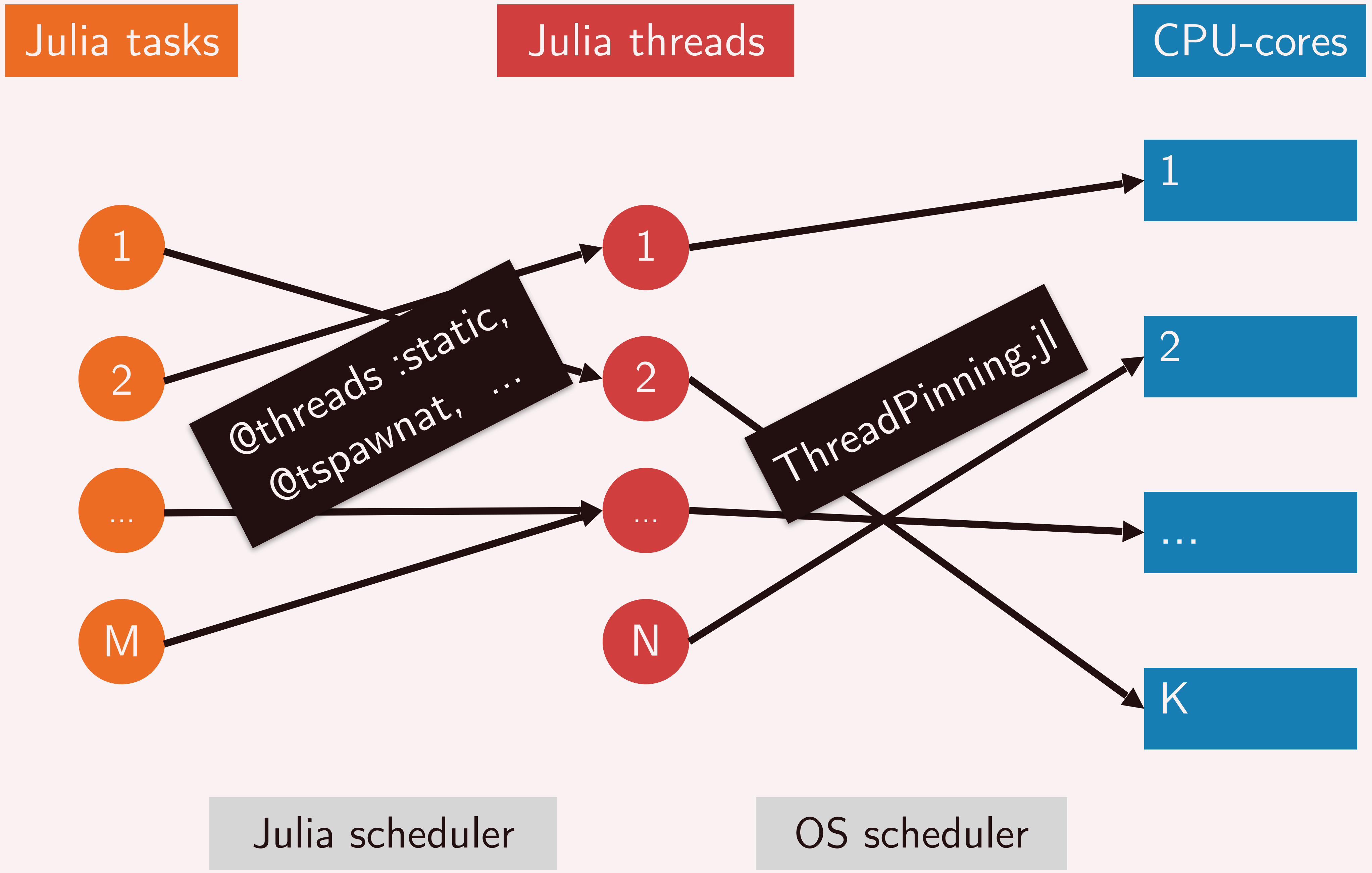
Task-based multithreading

M tasks on N threads

Why?

- Convenience
- Composability

You can opt out
(and may need to).



No easy way to produce (small) shared libraries.

PackageCompiler.jl is currently your best bet

Hampers integration into existing code bases

The Julia HPC Community

A small but vibrant and welcoming community.

People with passion and drive

International
(NERSC, ORNL, CSCS, PC2, ...)

Opportunity to join and grow

We welcome you to one of our sessions …

… or our monthly Zoom call

(open to everyone!)

Wrapping Up

Julia for HPC

Strengths

- Interactive and convenient
- Can be fast and scalable
- Inclusive and invites you to gradually delve deeper

Weaknesses

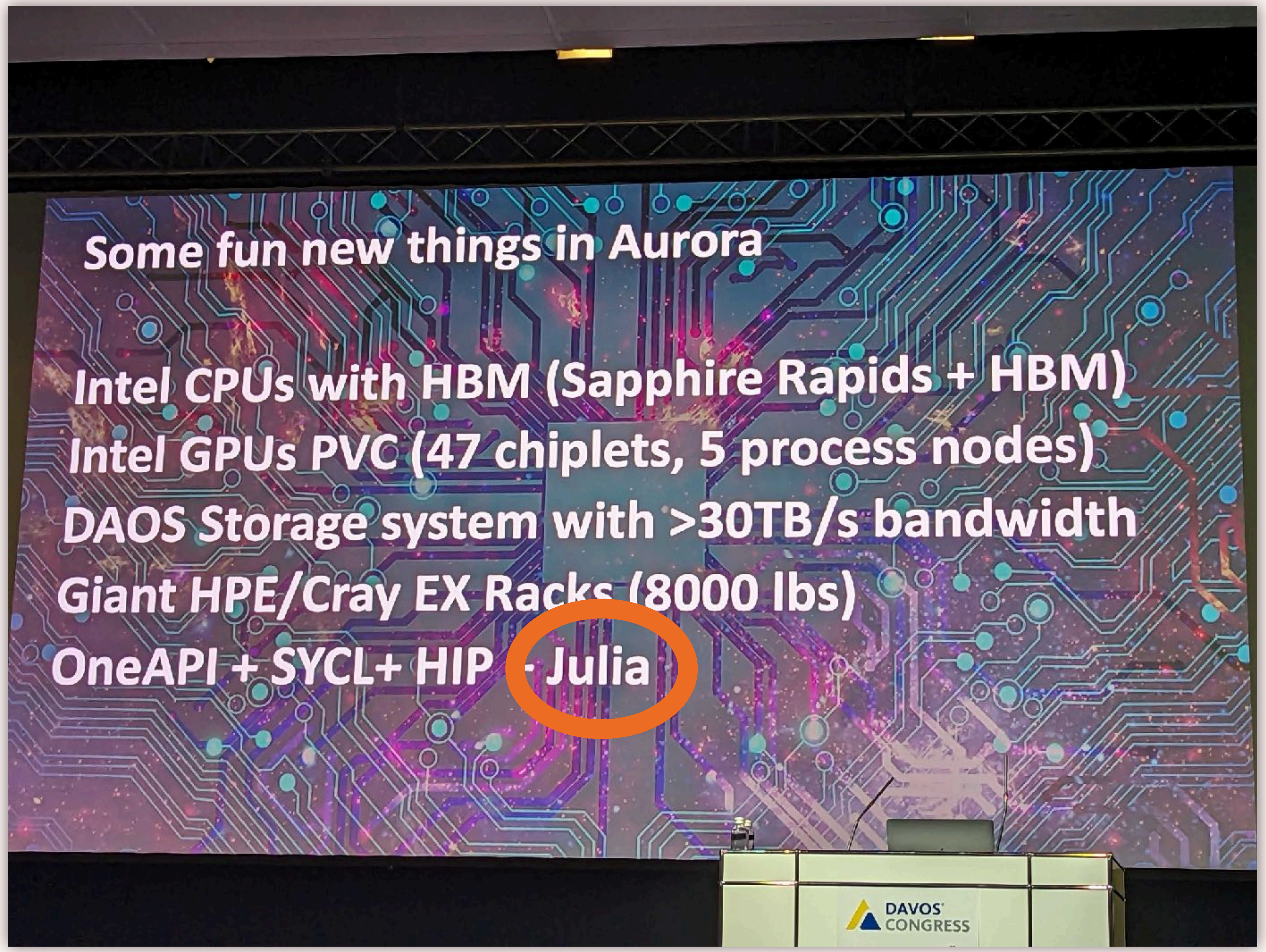
- Currently a maturing niche
- Achieving high performance can be tricky
- No easy way to produce (small) shared libraries.

Julia HPC Community

- Small but welcoming and vibrant

Julia has promising potential for HPC, and I invite you to join us in exploring and developing it.

Julia is a “fun new thing” on Aurora (ANL)



Invitation to read our overview paper

Churavy et al. (2022)

Journal Title
XX(X):1–14

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

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], 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's single-language paradigm, and its proven track record at achieving high-performance without sacrificing user productivity, makes it a viable single-language alternative to the existing composition of high-performance computing (HPC) languages (Fortran, C, C++) and higher-level languages (Python, R, Matlab) suitable for data analysis and simulation alike. Julia's rapid growth in language capabilities, package ecosystem, and community make it a promising new universal language for HPC similar to C++ or Python – an achievable goal if the community is given the necessary resources. This paper presents the views of a multidisciplinary group of researchers in academia, government, and industry advocating for the use of Julia and its ecosystem in HPC centers. We examine the current practice and role of Julia as a common programming model to address major challenges in scientific reproducibility, data-driven artificial intelligence/machine learning (AI/ML), co-design, and in-situ workflows, scalability and performance portability in heterogeneous computing, network, data management, and community education. As a result, we consider necessary the diversification of current investments to fulfill the needs of the upcoming decade as more supercomputing centers prepare for the Exascale era.

Keywords
High-Performance Computing, Julia, Programming Language