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"
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) ≈ 60k ranks
- ParallelStencil.jl (Multi-GPU) ≈ 1k GPUs
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

- \( x = \text{rand}(10000); \)

- \text{function} \ computation(x)\protect\footnote{The function computation(x) calculates flops per second and runtime.}
  - \( x = x^2 \)
  - \text{end;}

\textbf{Counted Flops:} 10000

### How?

- \text{using LIKWID}

- \text{metrics, events = @perfmon "FLOPS_DP" computation(x);}

\text{Compute from derived metrics}

\begin{verbatim}
10000
- begin
  - flops_per_second = metrics["FLOPS_DP"][1]["DP [MFLOP/s]"] * 1e6
  - runtime = metrics["FLOPS_DP"][1]["Runtime (RDTS) [s]"
  - flops = round(Int, flops_per_second * runtime)
  - end
\end{verbatim}
Threads can be *pinned* interactively

Pin the Julia threads

Visualize

```
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,
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,
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
```

# = 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, ...
```
Threads can be *pinned* interactively

Pin the Julia threads

```
julia> pinthreads(:numa)
```

```
julia> threadinfo(; groupby=:numa)
```

NUMA domains

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, |
| 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, |
| 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 |

# = 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

(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)
```

(≤ 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

- Code
  - @which
  - @less
  - @edit
  - Macro Exp.
    - @macroexpand
  - IR
    - @code_lowered
  - Typed IR
    - @code_warn_type
    - @code_typed
  - LLVM IR
    - @code_llvm
  - Assembly
    - @code_native
Insight into different code levels

Source code

```
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

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

Native code

```
@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
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)))
end
return y::Core.Const(2.23606797749979)
```

```julia
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.sqrt(5, 1.2, "3.0"::Vector{Any})
    (x = Main.rand(%1))
    (y = Main.sqrt(x))
end
return y
```

Random type!
Task-based multithreading

M tasks on N threads

Why?

‣ Convenience
‣ Composability

You can opt out
(and may need to).

@threads :static, @tspawnat, ...

ThreadPinning.jl

Julia tasks

CPU-cores

Julia threads

OS scheduler

Julia scheduler

@threads :static, @tspawnat, ...

ThreadPinning.jl

CPU-cores

OS scheduler

Julia scheduler

@threads :static, @tspawnat, ...

ThreadPinning.jl

CPU-cores

OS scheduler

Julia scheduler
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)

Some fun new things in Aurora

- Intel CPUs with HBM (Sapphire Rapids + HBM)
- Intel GPUs PVC (47 chiplets, 5 process nodes)
- DAOS Storage system with >30TB/s bandwidth
- Giant HPE/Cray EX Racks (8000 lbs)
- OneAPI + SYCL+ HIP + Julia
Bridging HPC Communities through the Julia Programming Language

Valentin Churavy¹, William F Godoy², Carsten Bauer³, Hendrik Ranocha⁴, Michael Schlottke-Lakemper⁵, Ludovic Räss⁶,⁷, Johannes Blaschke⁶, Mosè Giordano⁹, Erik Schnetter¹⁰,¹¹,¹², Samuel Omlin¹³, Jeffrey S. Vetter², Alan Edelman¹

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