Julia for Scientific Programmers

Tools for development scientific code for HPC applications in Julia

Lachlan Whyborn

September 29, 2023

The University of Queensland
Australia

What is Julia?

Programming language developed at the MIT Lincoln Laboratory, for the stated purpose of solving the ”two-language problem”. Started development in 2009, open beta in 2012, stable 1.0 release in 2018.

Why Julia?

Emphasis on *correctness*- almost impossible to create instances of undefined behaviour. Ideal for infrastructure software.

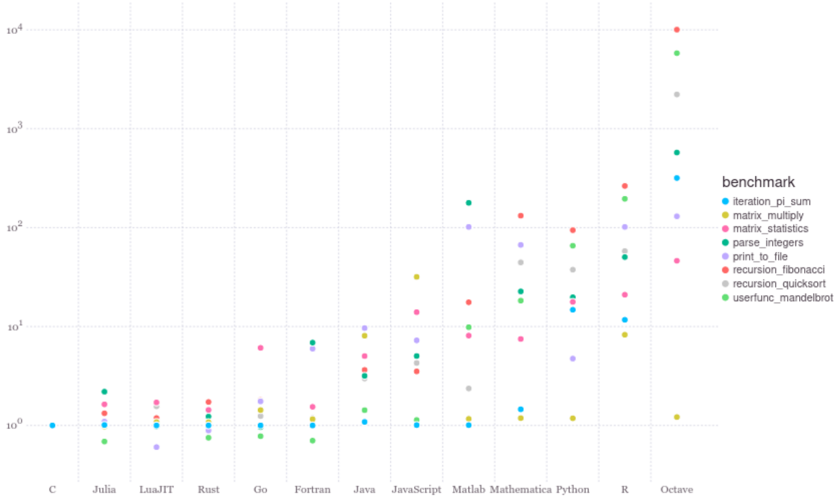
Designed for concurrency (not necessarily parallelism), targeting relatively light-weight and simple app development.

Why Julia?

Primary audience is scientific programmers/research software engineers- people writing software to solve engineering and scientific problems.

- Out-of-the-box parallelism, multi-thread and distributed.
- Multi-dimensional arrays and performant array operations as part of the standard library.
- Good package manager and ecosystem.
- REPL for fast exploration/debugging.

Some Benchmarks



Julia Usage

Some flagship users for the Julia language:

- NASA/JPL for spacecraft separation dynamics, data analysis of data-gathering missions.
- Brazilian space agency (INPE) for sattelite simulation and mission planning.
- CERN for analysis of data from the LHCb experiment.
- Climate Modeling Alliance (CliMA, led by CalTech) for global climate and weather modeling.
- Amazon for simulation of quantum computers.
- ASML for development of photolithographic machines for semiconductor manufacturing.

The Type System

Julia type system is dynamic, nominative and parametric.

- Dynamic: variable types are not fixed at compile-time, and are checked at run-time.
- Nominative: types are compared by name, not by structure.
- Parametric: code can take inputs of any type.

Concrete types in Julia are always final- one cannot inherit the *structure* of a type from another type. The type hierarchy still exists.

```julia
Float64 <: AbstractFloat <: Real <: Number <: Any  # true
```

Julia does not support typical "class" structures with methods out of the box.

Multiple Dispatch

Polymorphism is achieved through *multiple dispatch*. The correct function call is determined by the type of the inputs. Class methods are effectively *single dispatch*: They dispatch based on the type of the first argument. Multiple dispatch dispatches based on all arguments.

```julia
x = 1; y = 1.0; z = 1.0im

mul(a::AbstractInt, b::AbstractFloat) = a * b                         # Method 1
mul(a::AbstractInt, b::AbstractComplex) = a * b.re + (a * b.im)im    # Method 2
mul(a::Real, b::AbstractComplex) = a * b.re + (a * b.im)im           # Method 3

mul(x, y)   # Calls method 1
mul(y, z)   # Calls method 3
mul(x, z)   # AbstractInt subtypes Real, so calls method 2
```

Operator Broadcasting

Operations can be broadcast across collections using the "." operator. Looks similar to NumPy in Python.

```
x = LinRange(0, 10, 1001);
f = 2 * x.^2 + 3 * x .- 5     # Returns a 1001 length vector, but not efficient
↪   due to temporaries
f2 = @. 2 * x^2 + 3 * x - 5   # Operations are fused, does not create
↪   temporaries

y = reshape(collect(LinRange(0, 5, 200)), (1, 200)) # a (1, 200) matrix
f3 = @. 3 * x + 5 * y                               # Returns a (1001, 200)
↪   matrix
```

Multi-threaded and Distributed Support

Many options for multi-threaded and distributed operation. The most basic kind of parallelism, the isolated for loop:

```
func(x) = ... # some long running function
for val in vals          # Serial mode
    func(val)
end

using Distributed, Threads
@everywhere func(x) = ... # for distributed, all workers need a copy
(@distributed/Threads.@threads) for val in vals     # Parallel mode
    func(val)
end
```

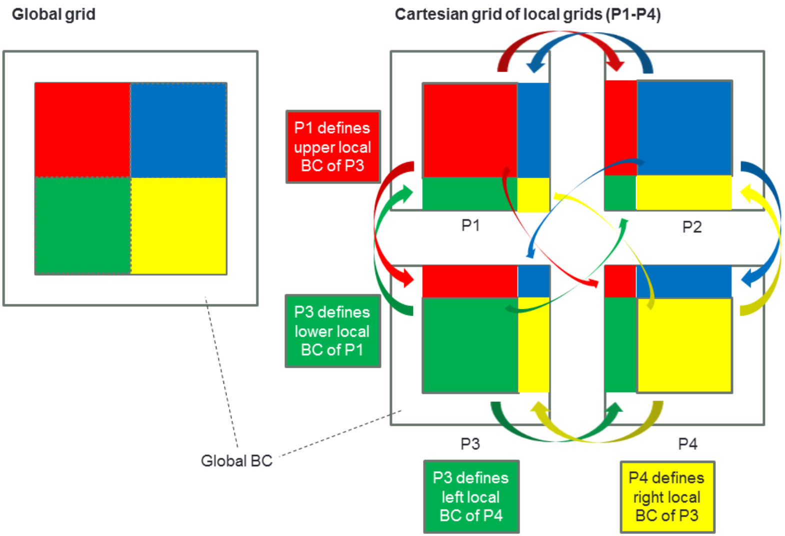
Other Tools in the Standard Library

- SharedArrays: For data shared across processes on a single host. All processes have access to the same information.
- pmap: For larger scale parallelism; the @distributed macro can handle both large and small scale parallelism relatively efficiently.
- Channels/Futures: tools for constructing data pipelines between processes.

Packages for HPC

- DistributedArrays.jl: Arrays which are distributed across processes.
- Polyester.jl: Lightweight multithreading package.
- Dagger.jl: Distributing tasks of various cost, inspired by Python's Dask.
- MPI.jl: Wrapper of the classic MPI framework.
- ImplicitGlobalGrid.jl: Easy support for partitioning of Cartesian meshes.

ImplicitGlobalGrid.jl



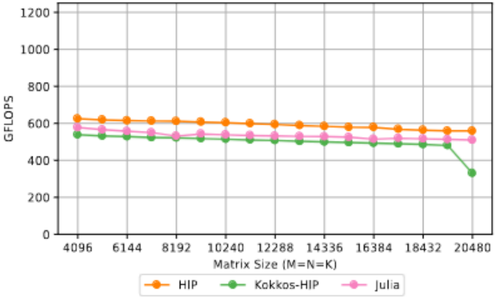
ImplicitGlobalGrid.jl

Exports three main functions:

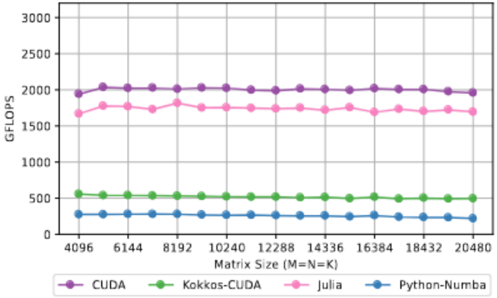
```
init_global_grid(nx, ny, nz)     # Init MPI setup
update_halo(A)                   # Updates halo (ghost) points for array A
finalize_global_grid()           # Gathers the grid to the master task
```

Also operates over many CUDA GPUs.

Strong support for CUDA through CUDA.jl, with reasonable support for AMD, oneAPI (Intel) and Metal (Apple).



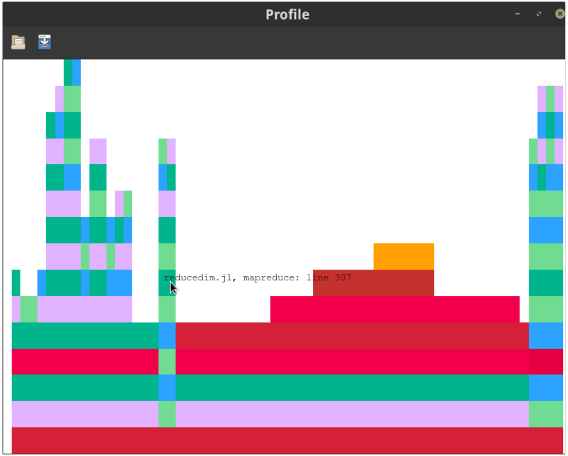
(a) Double precision (FP64)



(a) Double precision (FP64)

ProfView.jl

Starting point- generate a flamegraph for a function call.



Other Useful Debuggers

- Cthulhu.jl: Used to debug type inference issues.
- Infiltrator.jl: Inspect the stack in-situ.
- Debugger.jl: Typical debugger, breakpoints.

Modeling Toolkit

Julia's flagship modeling ecosystem. Machine learning, ODE and PDE systems, CAS and more. Solve a system similar to the Lorenz equations, with a second order term.

$$
\begin{aligned}
\frac{d^2x}{dt^2} &= \sigma(y - x) \\
\frac{dy}{dt} &= x(\rho - z) - y \\
\frac{dz}{dt} &= xy - \beta z
\end{aligned}
$$