Julia for HPC

Cédric Simal

cedric.simal@unamur.be

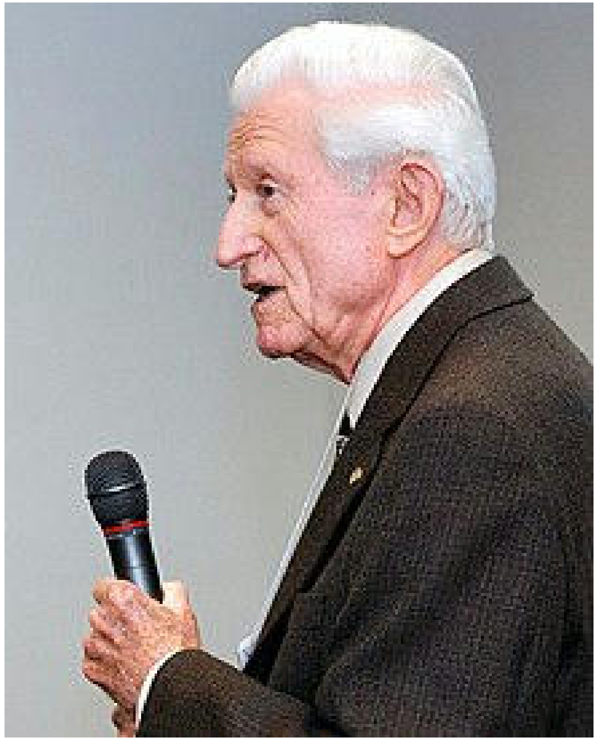
CéCI Training Sessions

16/11/2023

Science Anniversaries of today

Venera 3 Launch (1965)

Artemis 1 Launch (2022)



Birth of Gene Amdahl (1922)

image source: Wikipedia

Outline

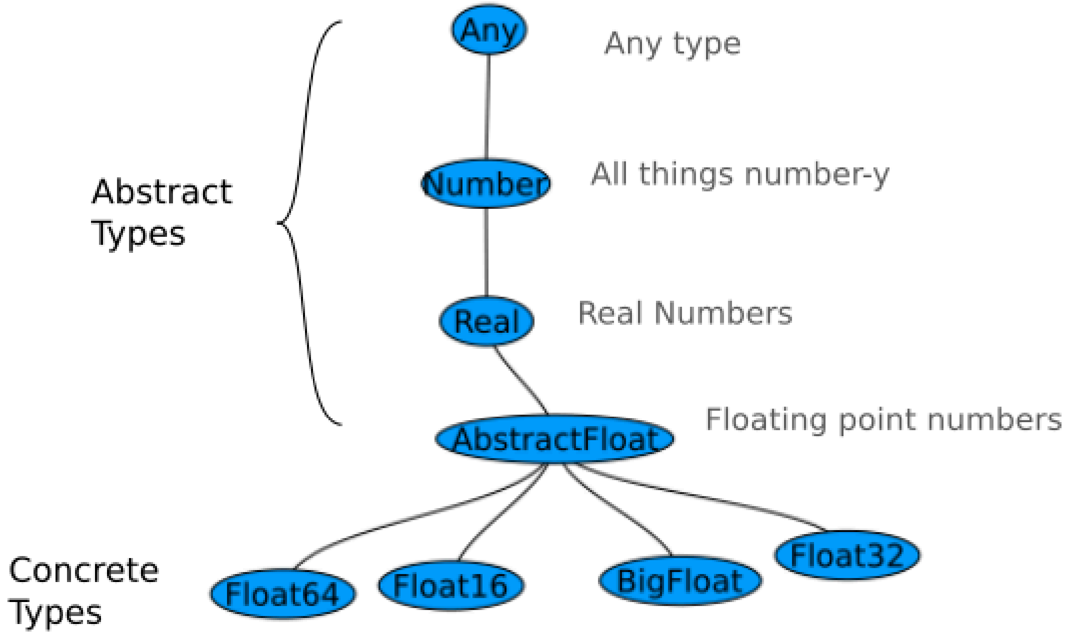
0. Recap on Multiple Dispatch
1. Benchmarking and Profiling Julia code
2. Performance Tips
3. Interop with other languages
4. Parallelism in the Standard library
5. Packages for HPC

Follow along!

https://github.com/csimal/CeCI-Julia-HPC

Recap on Multiple Dispatch

Abstract and Concrete Types



Multiple Dispatch

Functions can be overloaded based on the types of all their arguments

```julia
foo(x) = "That doesn't look like a number"
foo(x::Real) = "Got a real number"
foo(x::Integer) = "Got an integer"
foo(x::Int64) = "Got a 64 bit integer"
foo(x::AbstractFloat) = "Got a floating point number"
```

```julia
bar(x,y) = ""
bar(x::Real, y::Real) = "Two real numbers"
bar(x::T, y::T) where {T<:Real} = "Two real numbers of the same type"
bar(x::Integer, x::Integer) = "Two integers"
bar(x::Integer, y::AbstractFloat) = "An integer and a float"
```

Some gotchas with parametric types

```julia
struct Point2D{T}
    x::T
    y::T
end
```

```julia
# This will not work
function center(ps::Vector{Point2D{Real}})
    Point2D(
        mean(p.x for p in ps),
        mean(p.y for p in ps)
    )
end

# correct version
function center(ps::Vector{Point2D{<:Real}})
    Point2D(
        mean(p.x for p in ps),
        mean(p.y for p in ps)
    )
end
```

Some gotchas with parametric types

Explanation

```
Vector{Int} # a vector of Ints
Vector{T<:Real} where {T} # a vector of element type `T`, where `T` is a subtype of `Real`
Vector{T} where {T<:Real} # same as above
Vector{<:Real} # shorthand for above
Vector{Real} # a vector of mixed element types, all subtypes of `Real`
```

Benchmarking and Profiling

Benchmarking

Timing a single call

```
@time foo()
# See also
@timev foo() # verbose
@elapsed foo() # just the time
@allocated foo() # just the memory
used
```

For serious benchmarking

```
using BenchmarkTools

@benchmark foo()
# variants
@btime foo()
@belapsed foo()
@ballocated foo()
```

Profiling

Built-in Profiler

```julia
using Profile

@profile foo()
```

Visualizing profile data

```julia
using ProfileView

@profview foo()
```

```julia
using PProf

@pprof foo()
```

https://docs.julialang.org/en/v1.9/manual/profile/

Performance Tips

https://docs.julialang.org/en/v1.9/manual/performance-tips/

General Tips

- Put everything in functions
- Pre-allocate arrays, use in-place functions
- Global variables are evil. Use `const`
- Use multiple dispatch to break up functions
- Access Arrays column by column
- Abstract types at runtime are the devil

Array views

```
function foo!(a)
    for i in eachindex(a)
        a[i] = sin(a[i])
    end
end

# this won't work
for i in axes(A,1)
    foo!(A[i,:])
    # NB. taking the slice `A[i,:]`
creates a copy
end
```

Using views

```
for i in axes(A,1)
    foo!(view(A, i, :))
end

# convenient macro
@views for i in axes(A,1)
    foo!(A[i,:])
end
```

Type Stability

Don't do this...

```julia
relu(x) = x < 0 ? 0 : x

struct Foo
    x
    y::Real
    z::Vector{Integer}
end
```

but this

```julia
relu(x) = x < 0 ? zero(x) : x

struct Foo{T1,T2<:Real,T3<:Integer}
    x::T1
    y::T2
    z::Vector{T3}
end
```

Hint: Use the `@code_warntype` macro

Squeezing even more performance

- `StaticArray` (small arrays allocated on the Stack)
- `@inline` hint that a function should be inlined
- `@inbounds` disable bounds checking on array indexing
- `@fastmath` (floating point optimizations)
- `@simd` use Single Instruction Multiple Data CPU ops within a loop
- `PackageCompiler.jl` Create pre-compiled artefacts

Note: Use with caution!

https://juliaarrays.github.io/StaticArrays.jl/stable/

https://julialang.github.io/PackageCompiler.jl/dev/index.html

Interop with other languages

Calling Python/R

```julia
using PythonCall

re = pyimport("re")
words = re.findall("[a-zA-Z]+",
"PythonCall.jl is very useful!")
sentence = Py(" ").join(words)
pyconvert(String, sentence)
```

See also `PyCall.jl`

```julia
using Rcall

x = randn(10)
R"t.test($x)"

jmtcars = reval("mtcars");
rcall(:dim, jmtcars)
```

https://juliapy.github.io/PythonCall.jl/stable/

https://juliainterop.github.io/RCall.jl/stable/gettingstarted/

Parallelism in the Standard Library

Multi-threading

Start Julia with multiple threads

```
julia --threads=4
```

Checking number of threads/ thread id

```julia
Threads.nthreads()
Threads.threadid()
```

Running a loop over multiple threads

```julia
a = zeros(10)

Threads.@threads for i in eachindex(a)
    # your computation here
    a[i] = Threads.threadid()
end
```

https://docs.julialang.org/en/v1.9/manual/multi-threading/

Multithreading (2)

Data races

```julia
function sum_single(a)
    s = 0
    for i in a
        s += i
    end
    s
end
```

```julia
function sum_multi_bad(a)
    s = 0
    Threads.@threads for i in a
        s += i
    end
    s
end

function sum_multi_good(a)
    chunks = Iterators.partition(a, length(a) ÷ Threads.nthreads())
    tasks = map(chunks) do chunk
        Threads.@spawn sum_single(chunk)
    end
    chunk_sums = fetch.(tasks)
    return sum_single(chunk_sums)
end
```

Distributed Computing

Start Julia with multiple processors

```
julia -p 4
```

or

```julia
using Distributed
addprocs(4)
```

Low level interface

```julia
# call rand in processor 2 with argument (2,2)
r = remotecall(rand, 2, 2, 2)
# evaluate `1 .+ fetch(r)` in processor 2
s = @spawnat 2 1 .+ fetch(r)
# get the value of s
fetch(s)
# run expression on an arbitrary processor
t = @spawnat :any fetch(s)^2
```

https://docs.julialang.org/en/v1.9/manual/distributed-computing/

Loading dependencies accross processes

```julia
# define function on every processor
@everywhere function foo()
    ...
end
# load package
@everywhere using LinearAlgebra
# load Julia file
@everywhere include("utils.jl")
```

Launch processes with dependencies

```
julia -p <n> -L file1.jl -L file2.jl main.jl
```

Distributed Loops

```
count = 0
for i in 1:n
    count += (norm(rand(2)) < 1.0)
end
```

Distributed (map)reduce

```
count = @distributed (+) for i in 1:n
    Int(norm(rand(2)) < 1.0)
end
```

Can be applied to outside variables

```
a = rand(1000000)
@distributed (+) for i in eachindex(a)
    a[i]
end
```

Parallel map

```
pmap(collatz, 1:10000)
```

Shared Arrays

This will not work

```julia
a = zeros(1000)
@distributed for i in eachindex(a)
    a[i] = i
end
```

Use a `SharedArray`

```julia
using SharedArrays

a = SharedArray{Int}(1000)
@distributed for i in eachindex(a)
    a[i] = i
end
```

Intermezzo: Switching algorithms using Multiple Dispatch

```julia
function foo_single(n)
    for i in 1:n
        ...
    end
end

function foo_threaded(n)
    Threads.@threads for i in 1:n
        ...
    end
end

function foo_distributed(n)
    @distributed for i in 1:n
        ...
    end
end
```

Use empty types and multiple dispatch for user facing function

```julia
struct FooSingle end
struct FooThreaded end
struct FooDistributed end

foo(n) = foo(n, FooSingle())
foo(n, ::FooSingle) = foo_single(n)
foo(n, ::FooThreaded) = foo_threaded(n)
foo(n, ::FooDistributed) = foo_distributed(n)
```

Packages for HPC

Dagger.jl

```julia
@everywhere begin
    using Random
    Random.seed!(0)

    # Some "expensive" functions that complete at different speeds
    const crn = abs.(randn(20, 7))
    f(i) = sleep(crn[i, 7])
    g(i, j, y) = sleep(crn[i, j])
end
function nested_dagger()
    @sync for i in 1:20
        y = Dagger.@spawn f(i)
        for j in 1:6
            z = Dagger.@spawn g(i, j, y)
        end
    end
end
```

https://juliaparallel.org/Dagger.jl/dev/

Transducers.jl

Functional Style chains of operations

```
using Folds, Transducers

# runs in threaded mode by default
# NB. '▷' is the pipe operator
1:N ▷ Filter(isprime) ▷ Filter(ispalindromic) ▷ Map(n → 1) ▷ Folds.sum

# distributed fold
foldxd(+, (1 for n in 1:N if isprime(n) && ispalindromic(n)))
```

https://juliafolds.github.io/Transducers.jl/dev/

GPUs

Dedicated packages

- `CUDA.jl` (run Julia natively on Nvidia GPUs)
- `oneAPI.jl`
- `AMDGPU.jl` (ROCm)
- `Metal.jl` (MacOS - WIP)

High level libraries

- `Tullio.jl` (einsum operations)

CUDA kernel

```
function gpu_add!(y, x)
    index = (blockIdx().x - 1) * blockDim().x +
threadIdx().x
    stride = gridDim().x * blockDim().x
    for i = index:stride:length(y)
        @inbounds y[i] += x[i]
    end
    return
end

numblocks = ceil(Int, N/256)
fill!(y_d, 2)
@cuda threads=256 blocks=numblocks gpu_add3!(y_d,
x_d)
```

https://cuda.juliagpu.org/stable/tutorials/introduction/

Backend agnostic GPU kernels

```julia
using KernelAbstractions

@kernel function matmul_kernel!(a, b, c)
    i, j = @index(Global, NTuple)

    tmp_sum = zero(eltype(c))
    for k = 1:size(a)[2]
        tmp_sum += a[i,k] * b[k, j]
    end

    c[i,j] = tmp_sum
end
```

```julia
function matmul!(a, b, c)
    if size(a)[2] != size(b)[1]
        println("Matrix size mismatch!")
        return nothing
    end
    backend =
KernelAbstractions.get_backend(a)
    kernel! = matmul_kernel!(backend)
    kernel!(a, b, c, ndrange=size(c))
end
```

https://juliagpu.github.io/KernelAbstractions.jl/stable/examples/matmul/

https://b-fg.github.io/2023/05/07/waterlily-on-gpu.html

Honorable mentions

- `MPI.jl`
- `ClusterManagers.jl` Running slurm jobs directly from Julia
- `DrWatson.jl` Setting up scientific projects
- `libblastrampoline` Switch the BLAS/LAPACK backend dynamically at runtime