Introduction to the Julia language

Outline

- motivations
- julia as a numerical language
- types and methods
- about performance

Why yet another MATLAB like language ?

- *scripting languages* such as MATLAB, Scipy, Octave etc… are efficient to prototype algorithms (more than C++/Fortran)
- MATLAB is not free neither open source. It still remains the *lingua franca* in numerical algorithms.
- Octave is very slow (but highly compatible with MATLAB)
- Python along Numpy and Scipy is a beautiful language, but a little bit slow and do not support any parallelism as a built-in feature.
- accelerators such as Cython, Numba, Codon have pros/cons (what would be a long term choice ? do I have to learn a new syntax for different applications ?, …)
- R is well suited for statistics, but suffer from legacy syntax

Why do not try a new language for numerical computation ?

A language for numerical computation

- Julia's syntax is very similar to langages as MATLAB, Python or Scilab: moderate learning curve
- does not require *vectorized* code to run fast(JIT compiler)
- uses references (also for function arguments)
- indices start at `begin` (or `1`) and finish at `end` (akin to Fortran)
- use brackets `[i,j,...]` for indexing
- supports broadcasting (`A .= B .+ C .* D` or `@. A = B + C * D`)
- native nD arrays

Functional aspects

- anonymous functions : `(x -> x^2)(2)`
- closures : `f(x) = x + y`
- `map` , `reduce` , `filter` , ...
- variadic arguments, keyword argumets : `f(args...; kwargs...)`
- comprehension lists, generators
- convention used: in the `Base` module, function modifying their input arguments have an additional trailing `!` , e.g. inplace sort: `sort!`

Parallelism

Builtin - CPU

Built-in support for distributed (`Distributed`) and shared memory (`Threads`) parallelism:

```julia
using BenchmarkTools, Test

function add_seq!(y, x)
    for i ∈ eachindex(y, x)
        @inbounds y[i] += x[i]
    end
end

function add_threads_cpu!(y, x)  # JULIA_NUM_THREADS=4 or julia --threads=4
    Threads.@threads for i ∈ eachindex(y, x)
      @inbounds y[i] += x[i]
    end
end

function main(N = 2^20)
  x = fill(1.0f0, N)  # a vector filled with 1.0 (Float32)
  y = zeros(Float32, N)  # a vector filled with 0.0

  fill!(y, 2); add_seq!(y, x); @test all(==(3), y)
  @btime add_seq!($y, $x)
  # 158.857 μs (0 allocations: 0 bytes)

  fill!(y, 2); add_threads_cpu!(y, x); @test all(==(3), y)
  @btime add_threads_cpu!($y, $x)
  # 37.914 μs (24 allocations: 2.28 KiB)
end

main()
```

MPI - CPU

MPI.jl wraps `MPI` libraries for differents `ABI` s for classical two-sided communication for SPMD models.

```
# mpiexec -np 4 julia ex.jl
using MPI
MPI.Init()

function main(N = 4)
  comm = MPI.COMM_WORLD
  rank = MPI.Comm_rank(comm)
  size = MPI.Comm_size(comm)

  dst = mod(rank+1, size)
  src = mod(rank-1, size)

  send_mesg = Array{Float64}(undef, N)
  recv_mesg = Array{Float64}(undef, N)

  fill!(send_mesg, Float64(rank))

  rreq = MPI.Irecv!(recv_mesg, comm; source=src, tag=src+32)

  println("$rank: Sending   $rank -> $dst = $send_mesg")
  sreq = MPI.Isend(send_mesg, comm; dest=dst, tag=rank+32)

  MPI.Waitall([rreq, sreq])

  println("$rank: Received $src -> $rank = $recv_mesg")

  MPI.Barrier(comm)
end

main()
```

CUDA - GPU

CUDA.jl for GPU computations.

```julia
using CUDA

function add_seq_gpu!(y, x)
  for i ∈ 1:length(y)
    @inbounds y[i] += x[i]
  end
end

function bench_seq_gpu!(y, x)
  CUDA.@sync begin
    @cuda add_seq_gpu!(y, x)
  end
end

function add_threads_gpu!(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
end

function bench_threads_gpu!(y, x)
  threads = 256
  blocks = ceil(Int, length(y) / threads)
  CUDA.@sync begin
    @cuda threads=threads blocks=blocks add_threads_gpu!(y, x)
  end
end

function main(N = 2^20)
  x = CUDA.fill(1.0f0, N)  # a vector stored on the GPU filled with 1.0 (Float32)
  y = CUDA.fill(2.0f0, N)  # a vector stored on the GPU filled with 2.0

  fill!(y, 2); bench_seq_gpu!(y, x); @test all(==(3), Array(y))
  @btime bench_seq_gpu!($y, $x)
  # 61.511 ms (54 allocations: 3.33 KiB)

  fill!(y, 2); bench_threads_gpu!(y, x); @test all(==(3), Array(y))
  @btime bench_threads_gpu!($y, $x)
  # 63.579 µs (8 allocations: 400 bytes)
end

main()
```

External libraries calls

- sometimes you need to call a C/Fortran code
- "no boilerplate" philosophy : Julia does not require Mexfiles, Swig or other wrapping system
- code must be in a shared library
- various syntax:

```
ccall((:function, "lib"), return_type, (type_1,...,type_n), (arg_1, ..., arg_n))
@ccall "lib".function(arg_1::type_1, ..., arg_n::type_n)::return_type
```

- automatic generation of wrapper for C libraries (`Clang.jl`), lack of automatic C++ generators

Types

- there is a graph type in Julia reflecting the hierarchy of types: `Int64 <: Signed <: Integer <: Real <: Number <: Any`
- both abstract and concrete types are supported
- user can annotate the code with operator \texttt{::} "is an instance of"
- Julia supports
  - primitive types: `Int64, Float32, Bool, Char, ...`
  - composite types

```
struct Foo
  bar
  baz::Int32
  qux::Float64
end
```

- union types: `const OptionalInt = Union{Nothing,Int64}`
- parametric composite types (concrete or abstract):

```
struct Point{T}
  x::T
  y::T
end
```

- `Tuple`, `NamedTuple`, `VarArg` types
- type aliases
- function types

Multiple dispatch

- main idea : define piecewisely methods or functions depending on their arguments types
- let us define `f`

```
f(x::Float64, y::Float64) = 2x + y
f(x::Int, y::Int) = 2x + y
f(2., 3.) # returns 7.0
f(2, 3) # returns 7.0
f(2, 3.) # throws an ERROR: MethodError: no method matching f(::Int64, ::Float64)
```

- but if we define `g`

```
g(x::Number, y::Number) = 2x + y
g(2.0, 3) # now returns 7.0
```

- no automatic or magic conversions : for operators, arguments are promoted to a (user-definable) common type and use the specific implementation

- parametric methods

```julia
myappend(v::AbstractVector{T}, x::T) where {T <: Real} = [v..., x]
```

Performance (TODO)

- to prove that Julia is fast language, we did some tests
- benchmarks sources taken from the Julia site and modified

Conclusion

pros

- a true language, with lots of powerful features
- Julia is fast!

cons

- small community (package development)
- ...

not adressed

- meta-programming aspects : macros
- reflection

more information

- official website
- official forum
- documentation
- FAQ

Sources

- code examples inspired from CUDA docs