Julia: superglue for scientific computing

Joosep Pata
<pata@phys.ethz.ch>

ETH Zürich, Institute of Particle Physics

November 3, 2015

ETH Institute for
Particle Physics

Motivation

Restrict problem space to numerics - speed of C, simplicity of Python.

Optimizations based on existing technology - LLVM

Data analysis now common - benefit from external developments.

This talk is about how julia can be used for scientific computing.

Why another language?

The two-language problem

- static languages: great for experts, large, low-level applications, real-time
- data analysis and exploration: iteration, experimentation, unstructured

Data analysis now massively popular, a typical physicist has limited (quality) experience with C++: **just wants the result**(TM)

Enter Python, R: surging popularity in data analysis, science.

But **no** knowledge of types $\Rightarrow$ **no** fast machine code.

```
p = Particle()
foo(p) #must do explicit type checks every time
```

What has been tried?

Vectorize and offload ”heavy” stuff to a dedicated kernel?

- Leads to ”expert” and ”non-expert” code
- Artificial boundaries: user functions not callable
- Not every problem (easily) vectorizable: user-defined types
- Cost(human time) > Cost(machine time): but don't want to iterate weeks to try ideas!

Let the compiler do the hard work, write code as is most natural.

vectorized

```
rows = [row1, ...]
analyzeAll(rows)
```

serial

```
for (i=0;i<100;i++) {
  row = getRow(i);
  analyze(row);
}
```

The solution

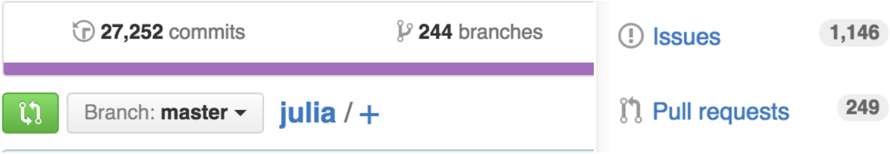
julia

High-level, fast, dynamically compiled numerics.

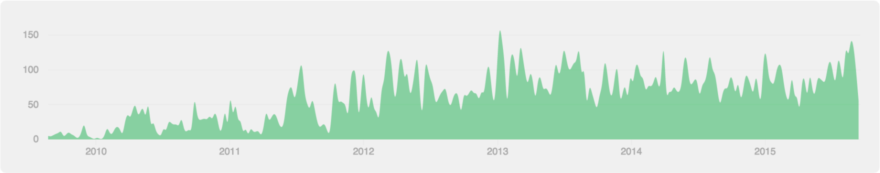
- Started at MIT CSAIL in 2011, now open-source, worldwide activity.
- Easy to use (like MATLAB, R), for generic numerical computing
- Used for physics, bio-informatics, statistics, image processing, finance
- Modular design: well-tested core + packages
- Code and issues tracked on github, (too) easy to contribute.
- Based on LLVM, OpenBLAS/Intel MKL

Development status

Core workflow and discussions on: github.com/JuliaLang/julia



Daily commit activity:



How it looks like...

Download binaries for OSX/Linux/Win, run REPL:

```
julia> 1+2
3
julia> Pkg.update()
julia> Pkg.add("Distributions")
julia> using Distributions, ROOT
julia> h = TH1D("h", "Hist", Int32(10), -10, 10)
```

- Run code in batch mode (e.g. on cluster): `julia code.jl`.
- Run interactive environment: `jupyter notebook`

The http://docs.julialang.org/en/release-0.4/manual/ is excellent.

Dynamic, optional types

Types may be specified or inferred automatically. User-side code auto-typed (time saver), library-side explicit-typed

```
julia> x = 1
1
julia> s = "asd"
"asd"
julia> bla = Uint32(2)
0x00000002
```

Operations with `x, s, bla` will be **machine-level**!
Functions may be typed, compiler figures out what to do at runtime:

```
julia> f(x::Int64) = x^2
julia> g(y, z) = sqrt(y^2 + z^2)
```

Multiple dispatch

- single dispatch: `spaceship.collide(asteroid)`
- multiple dispatch: `collide(spaceship, asteroid)`, `collide(spaceship, spaceship)`

Natural for mathematical code (operations are global). Easy to make common APIs by extending libraries. Leverages type system instead of boxing.

Underlying code

What code is actually generated for a function?

```
#define a new simple function
julia> f(x) = x > 0 ? -1 : 1
f (generic function with 2 methods)
#check produced code if x is Int64
julia> code_llvm(f, (Int64, ))
#that's the low-level LLVM code
define i64 @julia_f_21502(i64) {
top:
  %1 = icmp slt i64 %0, 1
  %. = select i1 %1, i64 1, i64 -1
  ret i64 %.
```

As fast as clang/C++!

Speed

fast basic functionality: loops, floating-point operations, external C/Fortran calls

Python:

```
In [11]: %time
for i in arange(100000000):
  x+=random.random()
Out [11] Wall time: 20.2 s
```

julia:

```
julia> @time (
for i=1:100000000
  x += rand()
end)
> 3.501108 seconds
```

Julia can outperform industry-standard C++ codes: tesselation.

Type system

Types are simple, low-overhead and fast! Can make types based on input data at runtime.

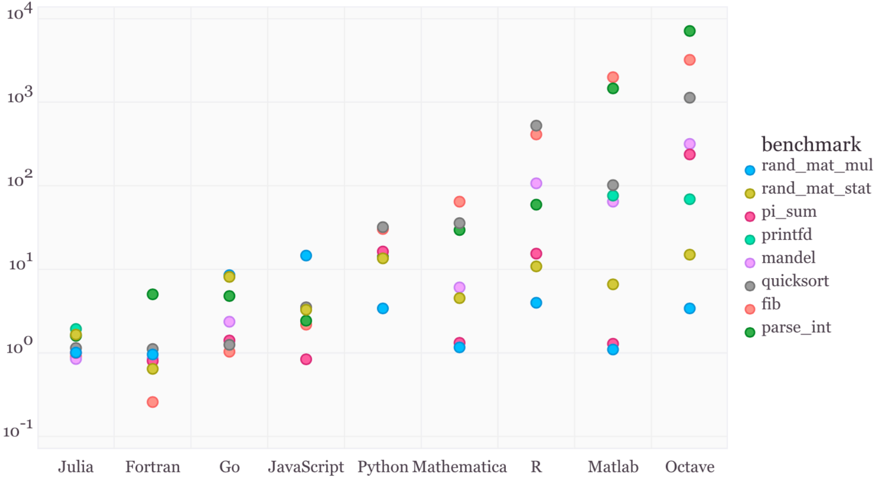
```
type Particle
  momentum::Float64 #explicitly specified
  name #no specified type, can be Anything
  friends::Vector{Particle} #complex type
end
my_p = Particle(0, "muon", [])
foo(p::Particle) = p.momentum^2
```

- Types are compiled: no overhead for e.g. `particle.momentum`
- code is specialized based on exact type specification information
- **no speed difference** with respect to built-in types
- types specify only data: multiple dispatch for member functions
- Easily add additional functions to types, e.g. `foo(t::TTree)`

Simple Mandlebrot example

```
function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1 #asd
        end
        z = z^2 + c
    end
    return maxiter
end
```

Speed comparisons



Naive julia implementation often similar to or better than C / Fortran,

Python interop

`PyCall.jl`, written by Steven Johnson (fftw)
Can import modules.

```
@pyimport numpy.random as nr
nr.rand(3,4)
```

Pass julia functions!

```
@pyimport scipy.optimize as so
so.newton(x -> cos(x) - x, 1)
```

Methods have special syntax: `my_dna.find("ACT")` becomes

```
@pyimport Bio.Seq as s
@pyimport Bio.Alphabet as a
my_dna = s.Seq("AGTACACTGGT", a.generic_dna)
my_dna[:find]("ACT")
```

C/Fortran interop

Can call C or Fortran libraries natively, no overhead.

```
const LIBOL = "libopenloops.so"
#id - process id (numeric)
#pp - array of particle momenta (4*N 1D)
#m2_tree - array with amplitude
function ol_evaluate_tree(id, pp, m2_tree)
  ccall(
      (:ol_evaluate_tree, LIBOL),
      Void,
      (Cint, Ptr{Cdouble}, Ptr{Cdouble}),
      Cint(id), pp, m2_tree
      )
end
ol_setparameter_int("order_ew", 1)
ol_evaluate_tree(id, pp, m2_tree)
```

C++ interop

From community package https://github.com/Keno/Cxx.jl, requires development version (0.5) of julia with latest (svn) LLVM.

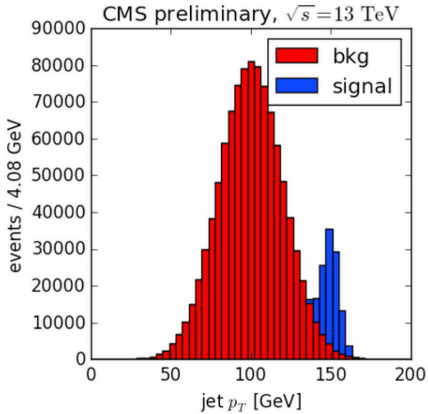
```
using Cxx
cxx""" #include<iostream> """

cxx"""
  float mycppfunction(int a, float b) {
    int y = (int)(b * 10.0);
    int x = 10;
    return a + x*y + 2;
}
"""
# Convert C++ to Julia function
julia> result = @cxx mycppfunction(1, 1.0)
```

Plotting

Many competing packages, matplotlib.pyplot very stable:

```
PyPlot.figure(figsize=(4, 4))
plot(h1, color="red", label="bkg");
plot(h2, color="blue", bottom=contents(h1), label="signal");
PyPlot.xlabel("jet \$p_T\$ [GeV]")
PyPlot.ylabel("events / $(round(diff(h1.edges[1])[1], 2)) GeV")
PyPlot.legend(loc="best")
PyPlot.title("CMS preliminary, \$\\sqrt{s} = 13\\ \\mathrm{TeV}\$")
```



One more thing...

- Support for automatic SIMD vectorization: `@simd for i=1:10000 x[i]*y[i] end` $\rightarrow$ linear speed-up
- Built-in parallelization, coroutines: one machine to a cluster, no `GIL` like in Python, memory-shared arrays
- Code $=$ Data: can manipulate program code on the fly (think LISP)
- Interactivity through **Jupyter** kernel, one of the main foci
- Language interop: C, Fortran natively, all of Python; soon C++ through add-ons: can wrap a complex library in a weekend.
- Many mature packages: statistics, storage, optimization, plotting, MVAs
- Soon: compile directly to GPU code

Notebook interface

Julia supported natively in the jupyter notebook.

- ROOT trees
- histograms
- statistics
- C++ interop

Traction

Number of active GitHub repositories

Language	Q2 2012	Q3 2014	% Growth
Julia	77	1,258	1,534%
C	24,080	64,597	168%
Java	50,334	175,968	250%
R	1,153	36,343	3,052%
Matlab	1,070	7,385	590%
Python	50,607	142,272	181%
GitHub Average	17,061	51,452	202%

Data source: GitHut

Summary

High-level code can still be fast.

Julia gaining traction in teaching, research and industry.

Julia used in and useful for high-energy physics

ROOT and julia are complementary.

Julia drawbacks

- Needs relatively new compiler (gcc 4.8+) and up-to-date software stack
- Less feature complete than python: database interface, domain-specific libraries
- garbage collection introduces real-time difficulties
- C++ support only in unstable development version with LLVM 3.8
-

Teaching

Julia is used in teaching @ MIT since 2013. Optimization, linear algebra, mathematical programming, numerical computation, PDEs.

- University of Edinburgh, Spring 2016
  - MATH11146, Modern optimization methods for big data problems (Prof. Peter Richtarik)
- MIT, Fall 2015
  - 6.251/15.081, Introduction to Mathematical Programming (Prof. Dimitris J. Bertsimas)
  - 18.06, Linear Algebra (Dr. Alex Townsend)
  - 18.303, Linear Partial Differential Equations: Analysis and Numerics (Prof. Steven G. Johnson)
  - 18.337/6.338, Numerical Computing with Julia (Prof. Alan Edelman). (IJulia notebooks)
  - 18.085/0851, Computational Science And Engineering I (Prof. Pedro J. Sáenz)
- "Sapienza" Univeristy of Rome, Italy, Spring 2015
  - Operations Research (Giampaolo Liuzzi)

Industry

Several companies are working with julia and contributing to it (open-source).

- Intel working on adding multi-threading to julia.
- Facebook working on database interfaces.
- Google investigating, some work on static analysis tools.
- Finance companies: BlackRock


Thank you! Questions?