High-performance scientific computing using julia

R³school/ELIXIR Training, Mirek Kratochvíl, Oliver Hunewald

April 22, 2021

High performance scientific computing using Julia

1. **Basics and motivation**
   Why would you want to choose Julia for your next project?
   Language primer and some distinctive features.
   Starting now, 45 minutes, followed by short Q&A and break (15 minutes)
2. **Scientific data processing**
   Tables, matrices, plots, files, …
   Start at 14:00 (UTC+02:00), 45 minutes, followed by Q&A + break
3. **Scaling your algorithms up**
   HPC, parallelization and distributed processing
   Start at 15:00 (UTC+02:00), 45 minutes, followed by Q&A, possibly problem-solving session

High performance scientific computing using Julia

1. **Basics and motivation**
   Why would you want to choose Julia for your next project?
   Language primer and some distinctive features.
   Starting now, 45 minutes, followed by short Q&A and break (15 minutes)
2. **Scientific data processing**
   Tables, matrices, plots, files, …
   Start at 14:00 (UTC+02:00), 45 minutes, followed by Q&A + break
3. **Scaling your algorithms up**
   HPC, parallelization and distributed processing
   Start at 15:00 (UTC+02:00), 45 minutes, followed by Q&A, possibly problem-solving session

High performance scientific computing using Julia

1. **Basics and motivation**
   Why would you want to choose Julia for your next project?
   Language primer and some distinctive features.
   Starting now, 45 minutes, followed by short Q&A and break (15 minutes)
2. **Scientific data processing**
   Tables, matrices, plots, files, ...
   Start at 14:00 (UTC+02:00), 45 minutes, followed by Q&A + break
3. **Scaling your algorithms up**
   HPC, parallelization and distributed processing
   Start at 15:00 (UTC+02:00), 45 minutes, followed by Q&A, possibly problem-solving session

Who’s talking?

Oliver Hunewald (LIH)

Miroslav Kratochvíl (LCSB Uni.lu)

Who's talking?

Oliver Hunewald (LIH)

Miroslav Kratochvíl (LCSB Uni.lu)

Fun fact: We met because of a Julia project!

Let’s start with some motivation

Wikipedia: Julia is a high-level, high-performance, dynamic programming language. While it is a general-purpose language and can be used to write any application, many of its features are well suited for numerical analysis and computational science. Distinctive aspects of Julia's design include a type system with parametric polymorphism in a dynamic programming language; with multiple dispatch as its core programming paradigm. Julia supports...

...but why?

The main distinctive features

Julia produces efficient programs.
Code is compiled to optimized machine representation before being executed. That means that your programs run very fast, you produce more results&insight in less time, and consume less energy.

The code is still very high-level.
You don't need to care about technical details as in other compiled languages. Most programs feel like in the usual scripting languages.

The ecosystem has a lot of goodies.
A great interpreter, easy distributed programming, wonderful modern packaging system, Jupyter notebooks.

Where does performance come from?

Program performance can be predicted quite precisely:

- Programs are evaluated by CPUs
- CPUs can hold a few numbers and execute instructions that modify them:
  - simple math
  - load a number from RAM, save a number to RAM
  - ...
- All instructions are predictably fast
  - adding 2 numbers usually takes 1 CPU cycle — in 2021, that's around 0.3ns
  - load/save takes a few cycles (from CPU cache) to 100's of cycles (from main memory)

Less instructions (and better cache utilization) translates to a faster program.

Where does performance come from?

Program performance can be predicted quite precisely:

- Programs are evaluated by CPUs
- CPUs can hold a few numbers and execute instructions that modify them:
  - simple math
  - load a number from RAM, save a number to RAM
  - ...
- All instructions are predictably fast
  - adding 2 numbers usually takes 1 CPU cycle — in 2021, that's around 0.3ns
  - load/save takes a few cycles (from CPU cache) to 100's of cycles (from main memory)

Less instructions (and better cache utilization) translates to a faster program.

Where does performance come from?

Program performance can be predicted quite precisely:

- Programs are evaluated by CPUs
- CPUs can hold a few numbers and execute instructions that modify them:
  - simple math
  - load a number from RAM, save a number to RAM
  - ...
- All instructions are predictably fast
  - adding 2 numbers usually takes 1 CPU cycle — in 2021, that's around 0.3ns
  - load/save takes a few cycles (from CPU cache) to 100's of cycles (from main memory)

Less instructions (and better cache utilization) translates to a faster program.

Where does performance come from?

Program performance can be predicted quite precisely:

- Programs are evaluated by CPUs
- CPUs can hold a few numbers and execute instructions that modify them:
  - simple math
  - load a number from RAM, save a number to RAM
  - ...
- All instructions are predictably fast
  - adding 2 numbers usually takes 1 CPU cycle — in 2021, that's around 0.3ns
  - load/save takes a few cycles (from CPU cache) to 100's of cycles (from main memory)

Less instructions (and better cache utilization) translates to a faster program.

Where does performance come from?

Program performance can be predicted quite precisely:

- Programs are evaluated by CPUs
- CPUs can hold a few numbers and execute instructions that modify them:
  - simple math
  - load a number from RAM, save a number to RAM
  - ...
- All instructions are predictably fast
  - adding 2 numbers usually takes 1 CPU cycle — in 2021, that's around 0.3ns
  - load/save takes a few cycles (from CPU cache) to 100's of cycles (from main memory)

Less instructions (and better cache utilization) translates to a faster program.

How to lose and save performance?

Python/R:

```
a + 1
```

How many instructions does this take?

How to lose and save performance?

Python/R:

a + 1

Computer:

1. Check if **a** exists in the available variables
2. Find address of **a**
3. Check if **a** is an actual object or null
4. Find if there is **__add__** in the object, get its address
5. Find if **__add__** is a function with 2 parameters
6. Load the value of **a**
7. Call the function, push Python call stack
8. Find if 1 is an integer and can be added
9. Check if **a** has a primitive representation (ie. not a big-int)
10. Run the primitive addition instruction (1 cycle!)
11. Pop Python call stack
12. Save the result to the place where Python can work with it

How to lose and save performance?

Python/R:

a + 1

...with static types

Computer:

1. Check if **a** exists in the available variables
2. Find address of **a**
3. Check if **a** is an actual object or null
4. Find if there is `__add__` in the object, get its address
5. Find if `__add__` is a function with 2 parameters
6. Load the value of **a**
7. Call the function, push Python call stack
8. Find if 1 is an integer and can be added
9. Check if **a** has a primitive representation (ie. not a big-int)
10. Run the primitive addition instruction (1 cycle!)
11. Pop Python call stack
12. Save the result to the place where Python can work with it

How to lose and save performance?

Python/R:

`a + 1`

…with static memory management

Computer:

1. Check if **a** exists in the available variables
2. Find address of **a**
3. Check if **a** is an actual object or null
4. Find if there is `__add__` in the object, get its address
5. Find if `__add__` is a function with 2 parameters
6. Load the value of **a**
7. Call the function, push Python call stack
8. Find if 1 is an integer and can be added
9. Check if **a** has a primitive representation (ie. not a big-int)
10. Run the primitive addition instruction (1 cycle!)
11. Pop Python call stack
12. Save the result to the place where Python can work with it

How to lose and save performance?

Python/R:

a + 1

...compiled with machine type support

Computer:

1. Check if **a** exists in the available variables
2. Find address of **a**
3. Check if **a** is an actual object or null
4. Find if there is `__add__` in the object, get its address
5. Find if `__add__` is a function with 2 parameters
6. Load the value of **a**
7. Call the function, push Python call stack
8. Find if 1 is an integer and can be added
9. Check if **a** has a primitive representation (ie. not a big-int)
10. Run the primitive addition instruction (1 cycle!)
11. Pop Python call stack
12. Save the result to the place where Python can work with it

How to lose and save performance?

Python/R:

a + 1

...with some compiler optimizations

Computer:

1. Check if **a** exists in the available variables
2. Find address of **a**
3. Check if **a** is an actual object or null
4. Find if there is `__add__` in the object, get its address
5. Find if `__add__` is a function with 2 parameters
6. Load the value of **a**
7. Call the function, push Python call stack
8. Find if 1 is an integer and can be added
9. Check if **a** has a primitive representation (ie. not a big-int)
10. Run the primitive addition instruction (1 cycle!)
11. Pop Python call stack
12. Save the result to the place where Python can work with it

Efficient = Fast

So how do I write efficient code?

Python / R

- Import a library written in another language
- Do not touch the data directly
- Only use library-defined API calls

```
import numpy as np

a = np.matrix([[1,2,3], [2,3,4], ...])
b = np.matrix([[2,3,4], ...])
c = np.matrix(
     np.array(a)*
     np.array(b))
```

So how do I write efficient code?

Python / R

- Import a library written in another language
- Do not touch the data directly
- Only use library-defined API calls

```
import numpy as np

a = np.matrix([[1,2,3], [2,3,4], ...])
b = np.matrix([[2,3,4], ...])
c = np.matrix(
     np.array(a)*
     np.array(b))
```

julia

- A precompiled, typed, partially staticized language
- Syntactic tools to make array processing easy
- Manual work with data is *not slow*

```
a = [ 1 2 3; 2 3 4; ...]
b = [ 2 3 4; ...]
c = a .* b
```

So how do I write efficient code?

Python / R

- Import a library written in another language
- Do not touch the data directly
- Only use library-defined API calls

```
import numpy as np

a = np.matrix([[1,2,3], [2,3,4], ...])
b = np.matrix([[2,3,4], ...])
c = np.matrix(
     np.array(a)*
     np.array(b))
```

julia

- A precompiled, typed, partially staticized language
- Syntactic tools to make array processing easy
- Manual work with data is *not slow*

```
a = [ 1 2 3; 2 3 4; ...]
b = [ 2 3 4; ...]
c = a .* b
```

Same performance:

```
c = zeros(size(a))
for i = 1:size(a, 1)
  for j = 1:size(a, 2)
    c[i,j] = a[i,j] * b[i,j]
  end
end
```

A canonical use-case

Let’s find if some sequences align well!

Catch: We’re scientists, we will very likely need to use a novel algorithm.

A canonical use-case

Let's find if some sequences align well!

Catch: We're scientists, we will very likely need to use a novel algorithm.

```
function LevenshteinDistance(char s[1..m], char t[1..n]):
  // for all i and j, d[i,j] will hold the Levenshtein distance between
  // the first i characters of s and the first j characters of t
  declare int d[0..m, 0..n]

  set each element in d to zero

  // source prefixes can be transformed into empty string by
  // dropping all characters
  for i from 1 to m:
      d[i, 0] := i

  // target prefixes can be reached from empty source prefix
  // by inserting every character
  for j from 1 to n:
      d[0, j] := j

  for j from 1 to n:
      for i from 1 to m:
          if s[i] = t[j]:
            substitutionCost := 0
          else:
            substitutionCost := 1

          d[i, j] := minimum(d[i-1, j] + 1,                   // deletion
                             d[i, j-1] + 1,                   // insertion
                             d[i-1, j-1] + substitutionCost)  // substitution

  return d[m, n]
```

Wikipedia: Levenshtein distance pseudocode

A canonical use-case

Let’s find if some sequences align well!

Catch: We’re scientists, we will very likely need to use a novel algorithm.

```
function levenshteinMatrix(s::Vector, t::Vector)
  m, n = length(s), length(t)
  d = zeros(Int, m+1, n+1)
  for i = 1:m
    d[i+1,1] = i
  end
  for i = 1:n
    d[1,i+1] = i
  end
  for i = 1:m
    for j = 1:n
      substCost = s[i]==t[i] ? 0 : 1
      d[i+1, j+1] =
        min(d[i, j+1] + 1,
            d[i+1, j] + 1,
            d[i, j] + substCost)
    end
  end
  return d
end
```

A canonical use-case

Let's find if some sequences align well!

Catch: We're scientists, we will very likely need to use a novel algorithm.

What if I try another language?

- C speedup: ≤1.5×
- R/Python slowdown: ≥50×

```
function levenshteinMatrix(s::Vector, t::Vector)
  m, n = length(s), length(t)
  d = zeros(Int, m+1, n+1)
  for i = 1:m
    d[i+1,1] = i
  end
  for i = 1:n
    d[1,i+1] = i
  end
  for i = 1:m
    for j = 1:n
      substCost = s[i]==t[i] ? 0 : 1
      d[i+1, j+1] =
        min(d[i, j+1] + 1,
            d[i+1, j] + 1,
            d[i, j] + substCost)
    end
  end
  return d
end
```

```
julia> levenshteinMatrix(collect("kitten"), collect("sitting"))
7×8 Array{Int64,2}:
 0  1  2  3  4  5  6  7
 1  1  2  3  4  5  6  7
 2  2  1  2  3  4  5  6
 3  3  2  1  2  3  4  5
 4  4  3  2  1  2  3  4
 5  5  4  3  2  2  3  4
 6  6  5  4  3  3  2  3
```

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

Starting up

Installation

Julia is available for most operating systems.
(We work regularly on Linuxes, Macs and Windows.)

- Linux:
  - `apt-get install julia`
  - `pacman install julia`
  - `emerge julia`
  - ...
- Mac&Windows: download from `julialang.org`

REPL

```
 ~ $ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.5.3
 _/ |\__'_|_|_|\__'_|  |  Debian ∴  julia/1.5.3+dfsg-3
|__/                   |

julia> 1+1
2

julia> 
```

REPL special modes

Extra function	hotkey	example
Package management	]	`add SomePackage`
Shell commands	;	`ls results/`
Help	?	`collect`
Completion	Tab	`is…?`

Basic language and syntax

Variables and expressions

Python

```
a = b + 123 / c
```

R

```
a <- b + 123 / c
```

julia

```
a = b + 123 / c
```

Printing out values

Python

```
print(123)
print("Hello!")
a = 23
print("A is now %d, twice A is %d"%(a, 2*a))
```

R

```
print(123)
print("Hello!")
a <- 23
print(paste0("A is now ", a, ", twice A is" , a*2))
```

julia

```
println(123)
println("Hello!")
a = 23
println("A is now $a, twice A is $(a*2)")

# or do it manually:
println("A is now " * string(a) *
        ", twice A is " * string(2*a))
```

Useful macros

```
@info "We're progressing!" a
```

...prints out:

```
┌ Info: We're progressing!
└   a = 23
```

```
@warn "Oh no, something looks bad" a
```

...prints out:

```
┌ Warning: Oh no, something looks bad
│   a = 23
└ @ Main REPL[3]:1
```

Structured data

- Tuples

```
(1, 5.23, "test")
```

Structured data

- Tuples

```
(1, 5.23, "test")
```

- Named tuples

```
(x=1, y=5.23, name="test")
```

Structured data

- Tuples

```
(1, 5.23, "test")
```

- Named tuples

```
(x=1, y=5.23, name="test")
```

- Structures

```
struct MyData
  x::Int
  y::Float64
  name::String
end
```

Structured data

- Tuples

```
(1, 5.23, "test")
```

- Named tuples

```
(x=1, y=5.23, name="test")
```

- Structures

```
struct MyData
  x::Int
  y::Float64
  name::String
end
```

- Constructors and accessors

```
a = MyData(2,3,"item")
a.x, a.name, ...
```

Arrays

```
vec = [1,2,3,4,5]
mtx = [1 2 3; 4 5 6; 0 0.123 0]

julia> mtx
3×3 Array{Float64,2}:
 1.0  2.0    3.0
 4.0  5.0    6.0
 0.0  0.123  0.0
```

Arrays

```
vec = [1,2,3,4,5]
mtx = [1 2 3; 4 5 6; 0 0.123 0]

julia> mtx
3×3 Array{Float64,2}:
 1.0  2.0    3.0
 4.0  5.0    6.0
 0.0  0.123  0.0
```

Arrays

```
vec = [1,2,3,4,5]
mtx = [1 2 3; 4 5 6; 0 0.123 0]

julia> mtx * mtx
3×3 Array{Float64,2}:
  9.0    12.369  15.0
 24.0    33.738  42.0
  0.492   0.615   0.738
```

Arrays

```
vec = [1,2,3,4,5]
mtx = [1 2 3; 4 5 6; 0 0.123 0]

julia> mtx .* mtx
3×3 Array{Float64,2}:
  1.0   4.0        9.0
 16.0  25.0       36.0
  0.0   0.015129   0.0
```

Arrays

```
vec = [1,2,3,4,5]
mtx = [1 2 3; 4 5 6; 0 0.123 0]

julia> vec[2]
2
```

Arrays

```
vec = [1,2,3,4,5]
mtx = [1 2 3; 4 5 6; 0 0.123 0]

julia> mtx[2,:]
3-element Array{Float64,1}:
 4.0
 5.0
 6.0
```

Arrays

```
vec = [1,2,3,4,5]
mtx = [1 2 3; 4 5 6; 0 0.123 0]

julia> mtx[:,2:3]
3×2 ArrayFloat64,2:
 2.0    3.0
 5.0    6.0
 0.123  0.0
```

Arrays

```
vec = [1,2,3,4,5]
mtx = [1 2 3; 4 5 6; 0 0.123 0]

julia> mtx[:,2]' * mtx[2,:]
33.738
```

Control structures — conditionals

Python

```
if a>=1:
  print("okay")
else:
  a = 1
```

R

```
if(a>=1)
  print("okay")
else
  a <- 1
```

julia

```
if a>=1
  println("okay")
else
  a = 1
end

# same:
if a>=1 println("okay")
else a = 1 end
```

Control structures — loops

Python

```python
for i in range(10):
  print(i)
```

R

```r
for(i in 1:10) print(i)
```

julia

```julia
for i in 1:10
  println(i)
end

# vectorized syntax:
println.(1:10);
```

Control structures — more loops

Python

```python
while i<5:
  i += 1
```

R

```r
while (i<5)
  i <- i+1
```

julia

```julia
while i<5
  i += 1
end
```

Function definitions

Python

```python
def myFunction(x, y)
  z = 2*x*y
  return (x+y+z)/3
```

R

```r
myFunction <- function(x, y) {
  z <- 2*x*y
  (x+y+z)/3
}
```

julia

```julia
function myFunction(x, y)
  z = 2*x*y
  return (x+y+z)/3
end
```

Function definitions

Python

```python
def myFunction(x, y)
  z = 2*x*y
  return (x+y+z)/3
```

R

```r
myFunction <- function(x, y) {
  z <- 2*x*y
  (x+y+z)/3
}
```

julia

```julia
function myFunction(x, y)
  z = 2*x*y
  return (x+y+z)/3
end

# shorter syntax:
myFunction(x, y) = (x+y+2*x*y)/3
```

Function definitions

Python

```python
def myFunction(x, y)
  z = 2*x*y
  return (x+y+z)/3
```

R

```r
myFunction <- function(x, y) {
  z <- 2*x*y
  (x+y+z)/3
}
```

julia

```julia
function myFunction(
  x::Number, y::Number)
  z = 2*x*y
  return (x+y+z)/3
end
```

Array machinery

Generating ranges with colons:

```
1:10        == [1, 2, 3, ..., 10]       # total 10
1:0.2:10    == [1, 1.2, 1.4, ..., 10]   # total 46
```

Array machinery

Generating ranges with colons:

```
1:10       == [1, 2, 3, ..., 10]      # total 10
1:0.2:10   == [1, 1.2, 1.4, ..., 10]  # total 46
```

Generating arrays:

```
[a^2 for a = 1:10]
```

Array machinery

Generating ranges with colons:

```
1:10       == [1, 2, 3, ..., 10]       # total 10
1:0.2:10   == [1, 1.2, 1.4, ..., 10]   # total 46
```

Generating arrays:

```
[a^2 for a = 1:10] == [1, 4, 9, 16, ..., 100]
```

...also `zeros`, `ones`, `fill`, `rand`, `randn`, `size`, `length`, `cat`, ...

Array machinery

Generating ranges with colons:

```
1:10       == [1, 2, 3, ..., 10]       # total 10
1:0.2:10   == [1, 1.2, 1.4, ..., 10]   # total 46
```

Generating arrays:

```
[a^2 for a = 1:10] == [1, 4, 9, 16, ..., 100]
```

...also `zeros`, `ones`, `fill`, `rand`, `randn`, `size`, `length`, `cat`, ...

Broadcasting over arrays:

```
1 .+ [1,2,3] == [2,3,4]
```

Array machinery

```
julia> vcat([1,2], [3,4])
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> hcat([1,2], [3,4])
2×2 Array{Int64,2}:
 1  3
 2  4
```

Array machinery

```
julia> vcat([1,2], [3,4])
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> hcat([1,2], [3,4])
2×2 Array{Int64,2}:
 1  3
 2  4
```

```
julia> a=[fill(i, (2,2)) for i in 1:5]
5-element Array{Array{Int64,2},1}:
 [1 1; 1 1]
 [2 2; 2 2]
 [3 3; 3 3]
 [4 4; 4 4]
 [5 5; 5 5]
```

Array machinery

```
julia> vcat([1,2], [3,4])
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> hcat([1,2], [3,4])
2×2 Array{Int64,2}:
 1  3
 2  4
```

```
julia> a=[fill(i, (2,2)) for i in 1:5]
julia> hcat(a...)
2×10 Array{Int64,2}:
 1  1  2  2  3  3  4  4  5  5
 1  1  2  2  3  3  4  4  5  5
```

Array machinery

```
julia> vcat([1,2], [3,4])
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> hcat([1,2], [3,4])
2×2 Array{Int64,2}:
 1  3
 2  4
```

```
julia> a=[fill(i, (2,2)) for i in 1:5]
julia> vcat(a...)
10×2 Array{Int64,2}:
 1  1
 1  1
 2  2
 2  2
 3  3
 3  3
 4  4
 4  4
 5  5
 5  5
```

Array machinery

```
julia> vcat([1,2], [3,4])
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> hcat([1,2], [3,4])
2×2 Array{Int64,2}:
 1  3
 2  4
```

```
julia> a=[fill(i, (2,2)) for i in 1:5]
julia> cat(dims=3, a...)
2×2×5 Array{Int64,3}:
[:, :, 1] =
 1  1
 1  1

[:, :, 2] =
 2  2
 2  2

 ⋮
```

Broadcasting over arrays

```
function.(array)
  == broadcast(function, array)
  == [function(a) for a in array]   # !
```

- Works with almost any function and operator
  `.+ .* ./ .+= .= .== .>= …`
- Makes the ‘trivial’ code much shorter (and less error-prone)
- Allows the compiler to reorder and parallelize the execution
- Often prevents creation of temporary arrays

Tricky question

What is the difference between

```
exp.(sin.(randn(10000000)))
```

and

```
[exp(x) for x in [sin(x) for x in randn(10000000)]]
```

?

Goodies!

Installing and using a package from the REPL

```
julia> using Pkg
julia> Pkg.add("Plots")
```

Loading the package:

```
julia> using Plots
```

Shortcut with `]`:

```
] add Plots
```

Installing and using a package from the REPL

```
julia> using Pkg
julia> Pkg.add("Plots")
```

Shortcut with `]`:

```
] add Plots
```

Loading the package:

```
julia> using Plots
julia> x=1:0.1:100

julia> @time plot(x, sin.(x) .* sin.(0.3 * x))
  4.599794 seconds (9.56 M allocations: 487.551 MiB, 3.02% gc time)
```

Installing and using a package from the REPL

```
julia> using Pkg
julia> Pkg.add("Plots")
```

Shortcut with]:

```
] add Plots
```

Loading the package:

```
julia> using Plots
julia> x=1:0.1:100

julia> @time plot(x, sin.(x) .* sin.(0.3 * x))
  4.599794 seconds (9.56 M allocations: 487.551 MiB, 3.02% gc time)

julia> @time plot(x, sin.(x) .* sin.(0.3 * x))
  0.001081 seconds (10.38 k allocations: 314.336 KiB)
```

Tricky question

Why was the first plot call so slow?

Trickier question

Why `3*1:10` works but `3*[1,2,3]` fails?

Benchmark everything!

```
julia> @time randn(10000,1000) * randn(1000,10000)
  4.287222 seconds (25 allocations: 915.547 MiB, 1.88% gc time)
```

Benchmark everything!

```
julia> @time randn(10000,1000) * randn(1000,10000)
  4.287222 seconds (25 allocations: 915.547 MiB, 1.88% gc time)

julia> @time randn(10000,1000) * randn(1000,10000)
  4.105074 seconds (6 allocations: 915.528 MiB, 0.13% gc time)
```

Benchmark everything!

```
julia> @time randn(10000,1000) * randn(1000,10000)
  4.287222 seconds (25 allocations: 915.547 MiB, 1.88% gc time)

julia> @time randn(10000,1000) * randn(1000,10000)
  4.105074 seconds (6 allocations: 915.528 MiB, 0.13% gc time)

julia> @time randn(1000,10000) * randn(10000,1000)
  0.451203 seconds (6 allocations: 160.218 MiB)
```

Make a UNIXy executable command-line tool

```julia
#!/usr/bin/env julia

using FileProcessor

if isempty(ARGS)
  @warn "No arguments, doing nothing!"
end

t = @timed for fn in ARGS
  @info "Processing file $fn"
  process_file(fn)
end

@info "Processing took $(t.time)s"
```

Make a UNIXy executable command-line tool

```julia
#!/usr/bin/env julia

using FileProcessor

if isempty(ARGS)
  @warn "No arguments, doing nothing!"
end

t = @timed for fn in ARGS
  @info "Processing file $fn"
  process_file(fn)
end

@info "Processing took $(t.time)s"
```

In console:

```
 $ chmod +x prog.jl
 $ ./prog.jl file1.csv file2.csv
[ Info: Processing file file1.csv
[ Info: Processing file file2.csv
[ Info: Processing took 0.054464386s
```

End of session 1

Q&A?

Takeaways:

- Julia is similar to many other languages, learning curve is very gentle
- Julia code is efficient by default
- Array broadcasting is a great way to write nice and fast code

High-performance scientific computing using julia

R³school/ELIXIR Training, Mirek Kratochvíl, Oliver Hunewald

April 22, 2021

High performance computing primer

Why do you need HPC?

The computation is too demanding even if done efficiently

\+ There is too much data to fit on a single computer

Why do you need HPC?

The computation is too demanding even if done efficiently

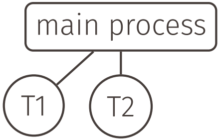
\+ There is too much data to fit on a single computer

= You need more computers

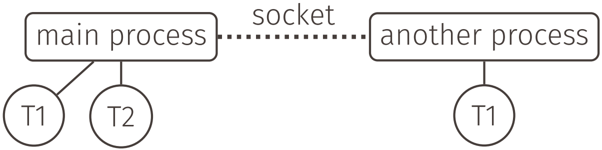
How does Julia help?

- The language is sufficiently high-level to abstract from the complexities of the distributed program execution
- There are great packages to help you
- Bonus: You don’t waste CPU cycles on many CPUs at once :)

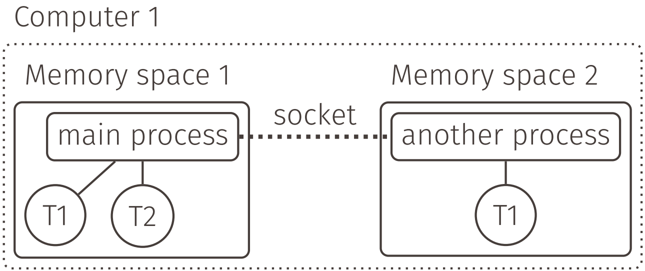
Julia model of parallel&distributed computation



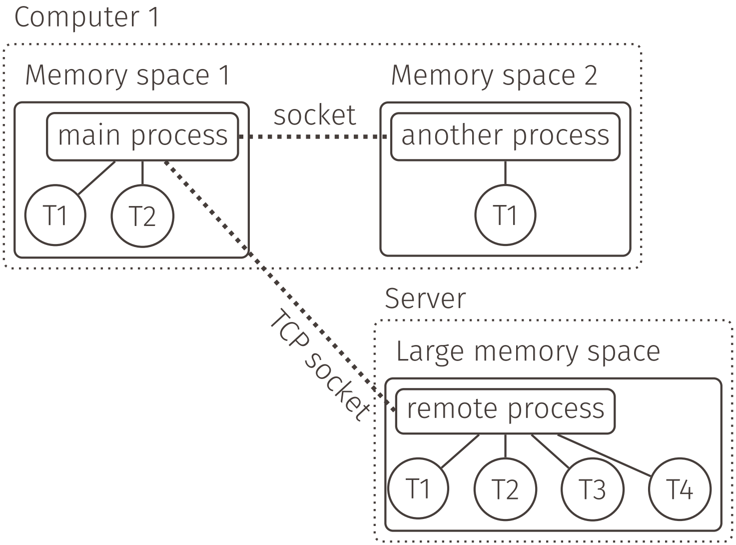
Julia model of parallel&distributed computation



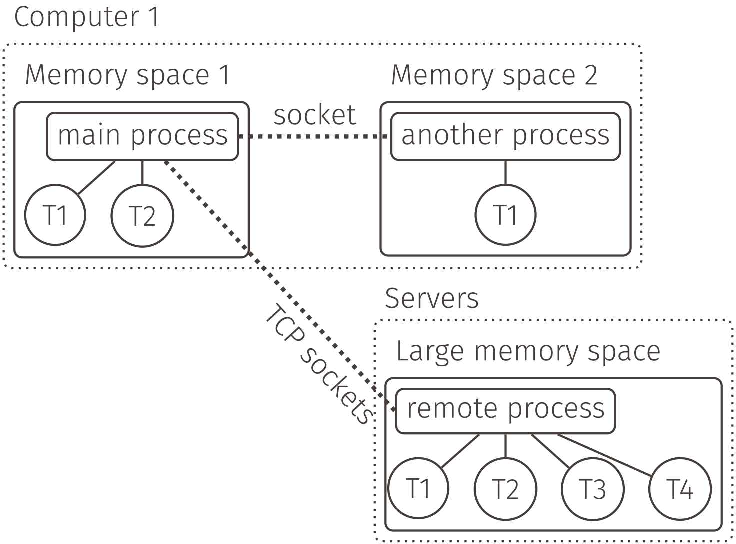
Julia model of parallel&distributed computation



Julia model of parallel&distributed computation



Julia model of parallel&distributed computation



Spawning a distributed process (locally)

```
julia> using Distributed
julia> addprocs(1)
```

Spawning a distributed process (remote one)

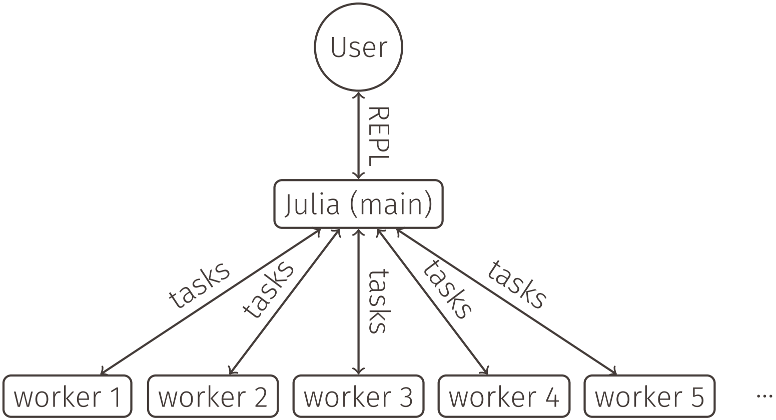
You need a working `ssh` connection to the server, ideally with keys:

```
user@pc> ssh server1
Last login: Wed Jan 13 15:29:34 2021 from 2001:a18:....
user@server> _
```

Spawning remote processes on remote machines:

```
julia> using Distributed
julia> addprocs([("server1", 10), ("pc2", 2)])
```

What do we have now?



Running something on the server

```
julia> workers()
4-element Array{Int64,1}:
 2
 3
 4
 5
julia>
```

Running something on the server

```
julia> workers()
4-element Array{Int64,1}:
 2
 3
 4
 5
julia> @everywhere using VeryHardComputationPackage
julia>
```

Running something on the server

```
julia> workers()
4-element Array{Int64,1}:
 2
 3
 4
 5
julia> @everywhere using VeryHardComputationPackage
julia> h = remotecall(() -> veryHardToComputeFunction(), 2)
Future(2, 1, 5, nothing)

julia>
```

Running something on the server

```
julia> workers()
4-element Array{Int64,1}:
 2
 3
 4
 5
julia> @everywhere using VeryHardComputationPackage
julia> h = remotecall(() -> veryHardToComputeFunction(), 2)
Future(2, 1, 5, nothing)

julia> fetch(h)
42
```

Doing it systematically

Doing it systematically

```
julia> a = randn(10000,10000)
julia> using DistributedArrays
julia> distribute(a)
julia> sum(a)
-7581.062238769015
```

…but all data still need to be loaded on a single computer?

Saving the memory

We made a simple wrapper for the distributed operations, now available in package `DistributedData`.

Idea: let's add some helpful syntactic sugar above the `remotecall`:

- `save_at(worker, :name, val)` evaluates `val` and saves it as a variable `name` on worker `worker`
- `get_from(worker, data)` fetches the data from the remote worker (returns a `Future`)

The following commands do *not* consume the precious memory on the main worker:

```
julia> save_at(1, :myData, :(randn(10000,10000)) )
julia> save_at(2, :myData, :(CSV.load("SuperHugeTable.csv")) )
```

The following commands do *not* consume the precious memory on the main worker:

```
julia> save_at(1, :myData, :(randn(10000,10000)) )
julia> save_at(2, :myData, :(CSV.load("SuperHugeTable.csv")) )
```

…what’s happening here?

Expressions

Julia code is a first class value, held in `Expression`s. You may *quote* the expressions to get them as values, and *evaluate* them elsewhere.

```
julia>
```

Expressions

Julia code is a first class value, held in `Expression`s. You may *quote* the expressions to get them as values, and *evaluate* them elsewhere.

```
julia> a=23
23

julia>
```

Expressions

Julia code is a first class value, held in `Expression`s. You may *quote* the expressions to get them as values, and *evaluate* them elsewhere.

```
julia> a=23
23

julia> a
23

julia>
```

Expressions

Julia code is a first class value, held in `Expression`s. You may *quote* the expressions to get them as values, and *evaluate* them elsewhere.

```
julia> a=23
23

julia> a
23

julia> :a
:a

julia>
```

Expressions

Julia code is a first class value, held in `Expression`s. You may *quote* the expressions to get them as values, and *evaluate* them elsewhere.

```
julia> a=23
23

julia> a
23

julia> :a
:a

julia> x=:(a+1)
:(a + 1)
```

```
julia>
```

Expressions

Julia code is a first class value, held in `Expression`s. You may *quote* the expressions to get them as values, and *evaluate* them elsewhere.

```
julia> a=23
23

julia> a
23

julia> :a
:a

julia> x=:(a+1)
:(a + 1)
```

```
julia> eval(x)
24

julia>
```

Expressions

Julia code is a first class value, held in `Expression`s. You may *quote* the expressions to get them as values, and *evaluate* them elsewhere.

```
julia> a=23
23

julia> a
23

julia> :a
:a

julia> x=:(a+1)
:(a + 1)
```

```
julia> eval(x)
24

julia> a=32
32

julia>
```

Expressions

Julia code is a first class value, held in `Expressions`. You may *quote* the expressions to get them as values, and *evaluate* them elsewhere.

```
julia> a=23
23

julia> a
23

julia> :a
:a

julia> x=:(a+1)
:(a + 1)
```

```
julia> eval(x)
24

julia> a=32
32

julia> eval(x)
33

julia>
```

Expressions

Julia code is a first class value, held in `Expression`s. You may *quote* the expressions to get them as values, and *evaluate* them elsewhere.

```
julia> a=23
23

julia> a
23

julia> :a
:a

julia> x=:(a+1)
:(a + 1)
```

```
julia> eval(x)
24

julia> a=32
32

julia> eval(x)
33

julia> :(a+$a+$x)
:(a + 32 + (a + 1))
```

Trick explanation

- `save_at(2, :myData, randn(10000,10000))`
  ...creates a 800MB random matrix in the main process, transfers it to worker 2, saves it in variable `myData` there
- `save_at(2, :myData, :(randn(10000,10000)))`
  ...creates a *tiny* expression, transfers it to worker 2, there it creates the random matrix and saves it in `myData`

Trick explanation

- `save_at(2, :myData, randn(10000,10000))`
  ...creates a 800MB random matrix in the main process, transfers it to worker 2, saves it in variable `myData` there
- `save_at(2, :myData, :(randn(10000,10000)))`
  ...creates a *tiny* expression, transfers it to worker 2, there it creates the random matrix and saves it in `myData`

Actual code:

```
save_at(worker, sym::Symbol, val) =
    remotecall(() -> Base.eval(Main, :(
        begin
            $sym = $val
            nothing
        end
    )), worker)
```

Trick explanation

- `save_at(2, :myData, randn(10000,10000))`
  ...creates a 800MB random matrix in the main process, transfers it to worker 2, saves it in variable `myData` there
- `save_at(2, :myData, :(randn(10000,10000)))`
  ...creates a *tiny* expression, transfers it to worker 2, there it creates the random matrix and saves it in `myData`

Actual code:

```
save_at(worker, sym::Symbol, val) =
    remotecall(() -> Base.eval(Main, :(
        begin
            $sym = $val
            nothing
        end
    )), worker)
```

This extends very easily

```
julia> r1 = get_from(3, :(veryHardFunction(dataPartOne)) )
julia> r2 = get_from(6, :(veryHardFunction(dataPartTwo)) )
```

...both workers are computing in parallel now!

This extends very easily

```
julia> r1 = get_from(3, :(veryHardFunction(dataPartOne)) )
julia> r2 = get_from(6, :(veryHardFunction(dataPartTwo)) )
```

...both workers are computing in parallel now!

```
julia> r = fetch(r1) + fetch(r2)
42
```

A bit of orchestration

```
julia> rs = [ get_from(w, :(
                findAlignmentScore(
                  FASTX.read("mySequence.fasta"),
                  FASTX.read("input$($i).fasta")
                )
              ))
              for (i,w) in enumerate(workers()) ]
```

…all workers are busy finding the alignments for your sequence now.

```
julia>
```

A bit of orchestration

```
julia> rs = [ get_from(w, :(
                findAlignmentScore(
                  FASTX.read("mySequence.fasta"),
                  FASTX.read("input$($i).fasta")
                )
              ))
              for (i,w) in enumerate(workers()) ]
```

...all workers are busy finding the alignments for your sequence now.

```
julia> fetch.(rs)
23-element ArrayFloat64,1:
 0.562544171906200‌5
 0.2189426530232181‌4
 0.90478880367169
 0.892124321016741‌8
 ⋮
```

Realistic examples

Use in a HPC environment (Slurm)

On the Slurm access node, create `script.jl`:

```
using Distributed
using ClusterManagers

n_workers = parse(Int, ENV["SLURM_NTASKS"])
addprocs_slurm(n_workers, topology=:master_worker)
⋮
```

Use in a HPC environment (Slurm)

On the Slurm access node, create `script.jl`:

```
using Distributed
using ClusterManagers

n_workers = parse(Int, ENV["SLURM_NTASKS"])
addprocs_slurm(n_workers, topology=:master_worker)
⋮
```

Execute on 1024 workers using:

```
srun -n 1024 -c 1 script.jl
```

The setup is similar for PBS and other HPC queueing systems.

Processing lots of independent tasks in parallel

```
files = readdir("myData/")

results = pmap(filename -> findSomeFeatures(filename),
               files,
               WorkerPool(workers()))
```

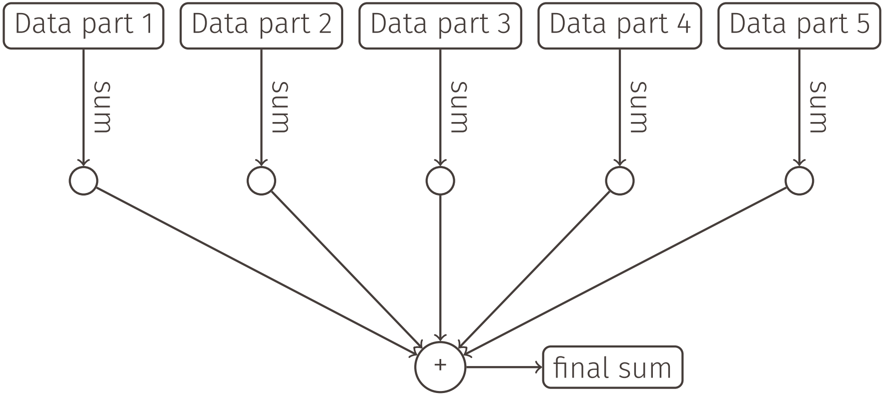
Working with partitioned datasets

Simplified example:

```
using DistributedData

scatter_array(:dataSlice, randn(1000000, 100), workers())
total = dmapreduce(:dataSlice, sum, +, workers())
unscatter(:dataSlice, workers())
```

Map/Reduce


Data part 1
Data part 2
Data part 3
Data part 4
Data part 5
sum
sum
sum
sum
sum
+
final sum

Working with partitioned datasets

```
using DistributedData

fetch.([save_at(w, :dataSlice, :(loadFile($fn)))
        for (w,fn) in zip(workers(), file_parts)])

result = dmapreduce(:dataSlice,
                    veryHardFunction,
                    (a,b)->combineSubresults(a,b),
                    workers())
```

Working with partitioned datasets

```
using DistributedData

fetch.([save_at(w, :dataSlice, :(loadFile($fn)))
        for (w,fn) in zip(workers(), file_parts)])

result = dmapreduce(:dataSlice,
                    veryHardFunction,
                    (a,b)->combineSubresults(a,b),
                    workers())
```

Working with partitioned datasets

```
using DistributedData

fetch.([save_at(w, :dataSlice, :(loadFile($fn)))
        for (w,fn) in zip(workers(), file_parts)])

result = dmapreduce(:dataSlice,
                    veryHardFunction,
                    (a,b)->combineSubresults(a,b),
                    workers())
```

Working with partitioned datasets

```
using DistributedData

fetch.([save_at(w, :dataSlice, :(loadFile($fn)))
        for (w,fn) in zip(workers(), file_parts))

result = dmapreduce(:dataSlice,
                    veryHardFunction,
                    (a,b)->combineSubresults(a,b),
                    workers())
```

Working with partitioned datasets

```
using DistributedData

fetch.([save_at(w, :dataSlice, :(loadFile($fn)))
        for (w,fn) in zip(workers(), file_parts)])

result = dmapreduce(:dataSlice,
                    veryHardFunction,
                    (a,b)->combineSubresults(a,b),
                    workers())
```

Working with partitioned datasets

```
using DistributedData

fetch.([save_at(w, :dataSlice, :(loadFile($fn)))
        for (w,fn) in zip(workers(), file_parts)])

result = dmapreduce(:dataSlice,
                    veryHardFunction,
                    (a,b)->combineSubresults(a,b),
                    workers())
```

Working with partitioned datasets

```
using DistributedData

fetch.([save_at(w, :dataSlice, :(loadFile($fn)))
        for (w,fn) in zip(workers(), file_parts)])

result = dmapreduce(:dataSlice,
                    veryHardFunction,
                    (a,b)->combineSubresults(a,b),
                    workers())
```

Working with partitioned datasets

```
using DistributedData

fetch.([save_at(w, :dataSlice, :(loadFile($fn)))
        for (w,fn) in zip(workers(), file_parts)])

result = dmapreduce(:dataSlice,
                    veryHardFunction,
                    (a,b)->combineSubresults(a,b),
                    workers())
```

Working with partitioned datasets

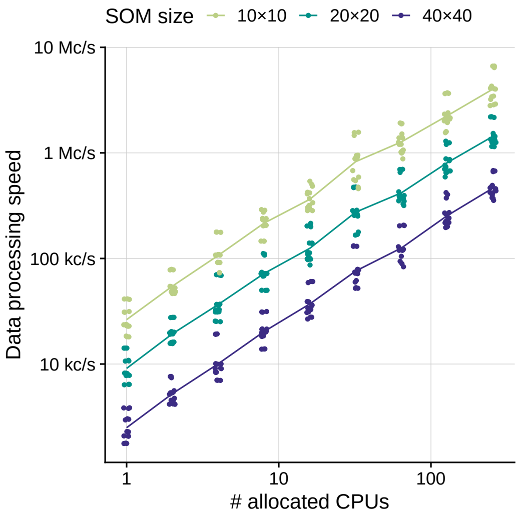
```
using DistributedData

fetch.([save_at(w, :dataSlice, :(loadFile($fn)))
        for (w,fn) in zip(workers(), file_parts)])

result = dmapreduce(:dataSlice,
                    veryHardFunction,
                    (a,b)->combineSubresults(a,b),
                    workers())
```

Tricky question: why `fetch`?

Scaling up



SOM size
10×10
20×20
40×40
10 Mc/s
1 Mc/s
100 kc/s
10 kc/s
Data processing speed
1
10
100
allocated CPUs

Where to go next?

Where to go next?

Julia packages make great building blocks:

- `JuMP.jl` — linear optimization
- `Gen.jl`, `Distributions.jl` — probabilistic programming and statistics
- `Knet.jl`, `Mocha.jl` — deep learning
- `DifferentialEquations.jl` — ODE solving
- major bioinformatics formats can be opened easily

$\vdots$

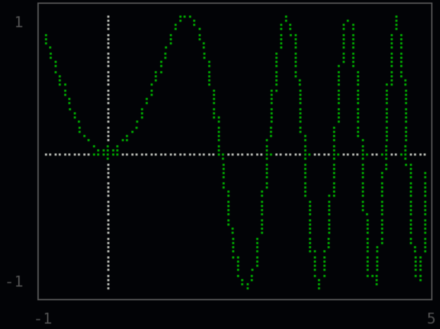
Cool language features we did not cover:

- Overloading based on multiple dispatch
- Rich type system with subtypes

Goodies everywhere

```
julia> using UnicodePlots

julia> lineplot(-1:0.1:5, sin.(collect(-1:0.1:5).^2))
```




```
julia>
```

Optimization starts here!

```
function sum1(d::Matrix)
    n,m = size(d)
    s = 0
    for i = 1:n
        for j = 1:m
            s += d[i,j]
        end
    end
    return s
end
```

Optimization starts here!

```
function sum1(d::Matrix)
    n,m = size(d)
    s = 0
    for i = 1:n
        for j = 1:m
            s += d[i,j]
        end
    end
    return s
end
```

```
julia> mydata = randn(1000,100000)

julia> @time sum1(mydata)
  0.524455 seconds (1 allocation: 16 bytes)
-3802.7179206

julia> @time sum1(mydata)
  0.541449 seconds (1 allocation: 16 bytes)
-3802.7179206
```

Optimization starts here!

```julia
function sum1(d::Matrix)
    n,m = size(d)
    s = 0
    for i = 1:n
        for j = 1:m
            s += d[i,j]
        end
    end
    return s
end

function sum2(d::Matrix)
    n,m = size(d)
    s = 0
    for j = 1:m
        for i = 1:n
            s += d[i,j]
        end
    end
    return s
end
```

```
julia> mydata = randn(1000,100000)

julia> @time sum1(mydata)
  0.524455 seconds (1 allocation: 16 bytes)
-3802.7179206

julia> @time sum1(mydata)
  0.541449 seconds (1 allocation: 16 bytes)
-3802.7179206
```

Optimization starts here!

```
function sum1(d::Matrix)
    n,m = size(d)
    s = 0
    for i = 1:n
        for j = 1:m
            s += d[i,j]
        end
    end
    return s
end

function sum2(d::Matrix)
    n,m = size(d)
    s = 0
    for j = 1:m
        for i = 1:n
            s += d[i,j]
        end
    end
    return s
end
```

```
julia> mydata = randn(1000,100000)

julia> @time sum1(mydata)
  0.524455 seconds (1 allocation: 16 bytes)
-3802.7179206

julia> @time sum1(mydata)
  0.541449 seconds (1 allocation: 16 bytes)
-3802.7179206

julia> @time sum2(mydata)
  0.103912 seconds (1 allocation: 16 bytes)
-3802.7179206

julia> @time sum2(mydata)
  0.103962 seconds (1 allocation: 16 bytes)
-3802.7179206
```

Takeaways

- You don't need C and C++ to write super-fast code
- Julia ecosystem provides lots of functionality for easy data processing
- Writing distributed code does not require learning MPI, OpenMP, Spark, ...
- Efficient code is faster and generates more results :)

Thank you for attention!

Q&A?

Please help us make the courses better!

Complete the survey at:

`https://is.gd/Juliaelixir202104`