High-performance scientific computing using julia

R^{3}school/ELIXIR Training, Mirek Kratochvíl, Oliver Hunewald

April 22, 2021

Course materials

https://elixir-luxembourg.github.io/julia-training/

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

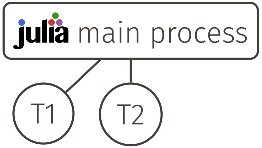
\+ There is too much data to fit on a single computer

= You need more computers

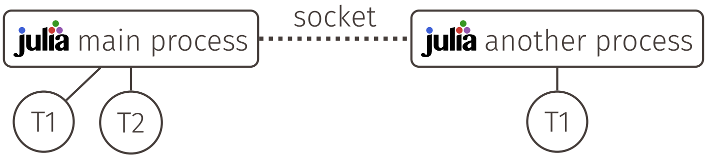
How does Julia help?

- 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 :)

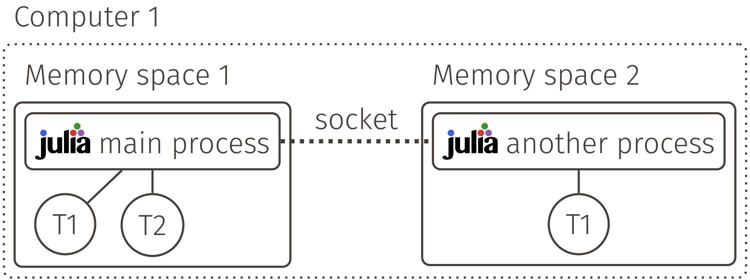
Julia model of parallel&distributed computation



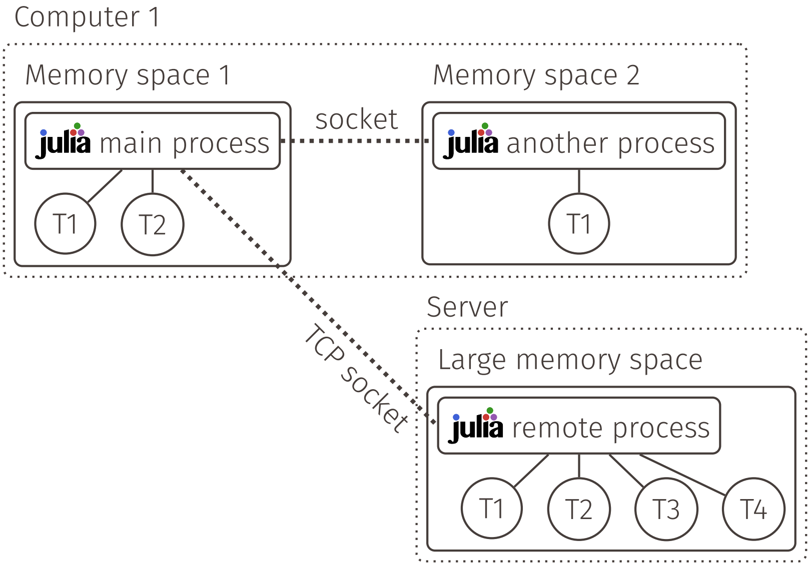
Julia model of parallel&distributed computation



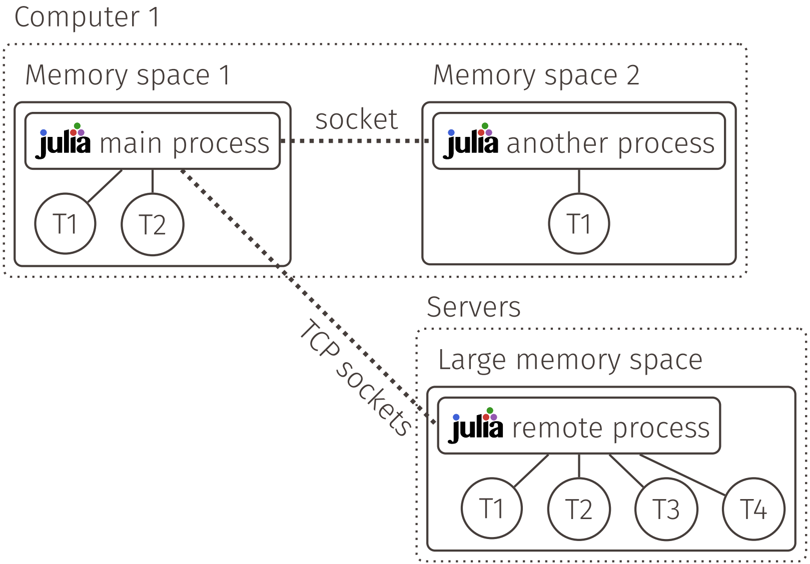
Julia model of parallel&distributed computation



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)

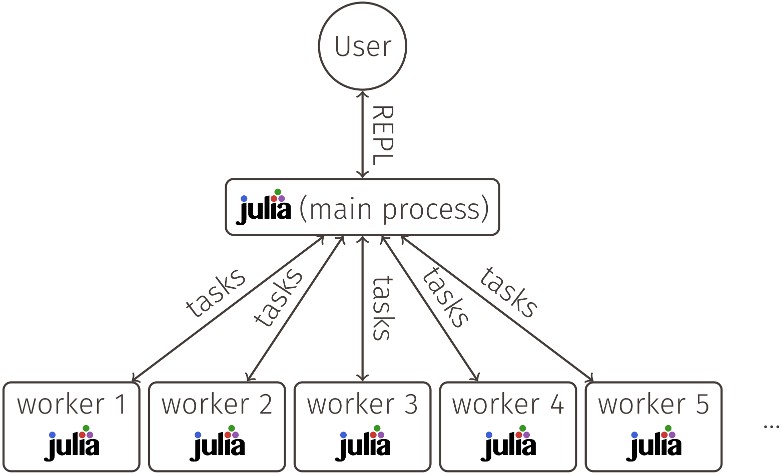
You need a working `ssh` connection to the server, ideally with keys:

```
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 around 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(6, :myData, :(randn(10000,10000)) )
julia> save_at(4, :myData, :(CSV.load("SuperHugeTable.csv")) )
```

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

```
julia> save_at(6, :myData, :(randn(10000,10000)) )
julia> save_at(4, :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 `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>
```

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>
```

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)
```

Getting the data back

- `get_from(2, :myData)`
  ...evaluates `myData` on the remote worker and returns whatever was evaluated as a `Future`
- `get_from(2, :(sum(myData)))`
  ...evaluates the function (`sum(myData)`) on the remote worker and returns the result as a `Future`

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:
 0.562544171906200 5
 0.2189426530232181 4
 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 julia 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()))
```

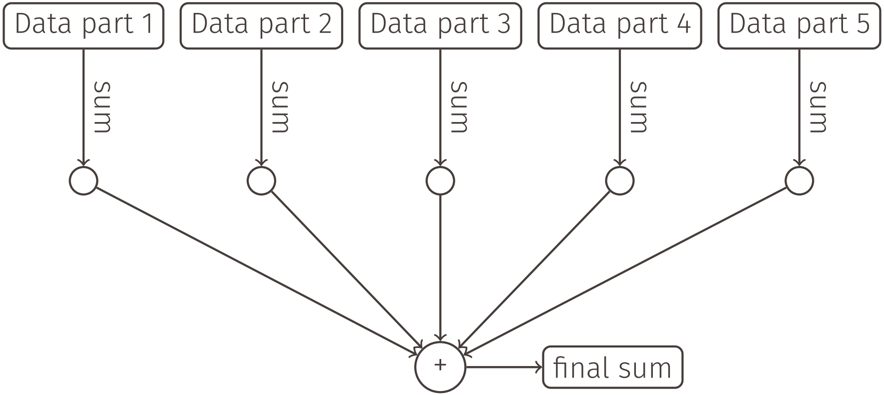
Working with partitioned datasets

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

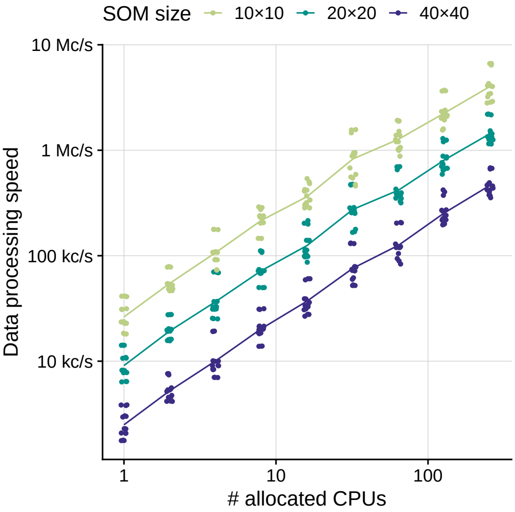
```
using DistributedData

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



https://github.com/LCSB-BioCore/GigaSOM.jl

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

$\vdots$

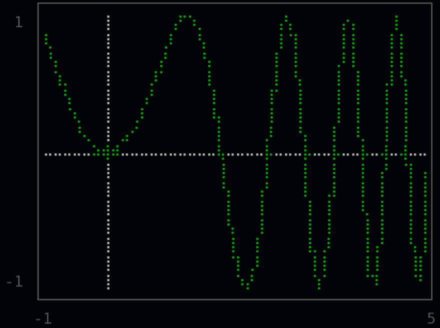
Cool language features we did not cover:

- 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!

```
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
```

Optimization tips

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

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?

miroslav.kratochvil@uni.lu
oliver.hunewald@lih.lu

Please help us make the courses better!

Complete the survey at:

https://is.gd/Juliaelixir202104