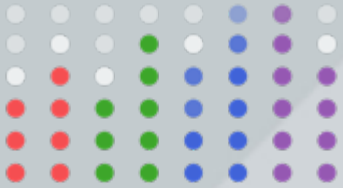


julia

Parallel Computing's Biggest Challenge

CSAIL

Julia computing

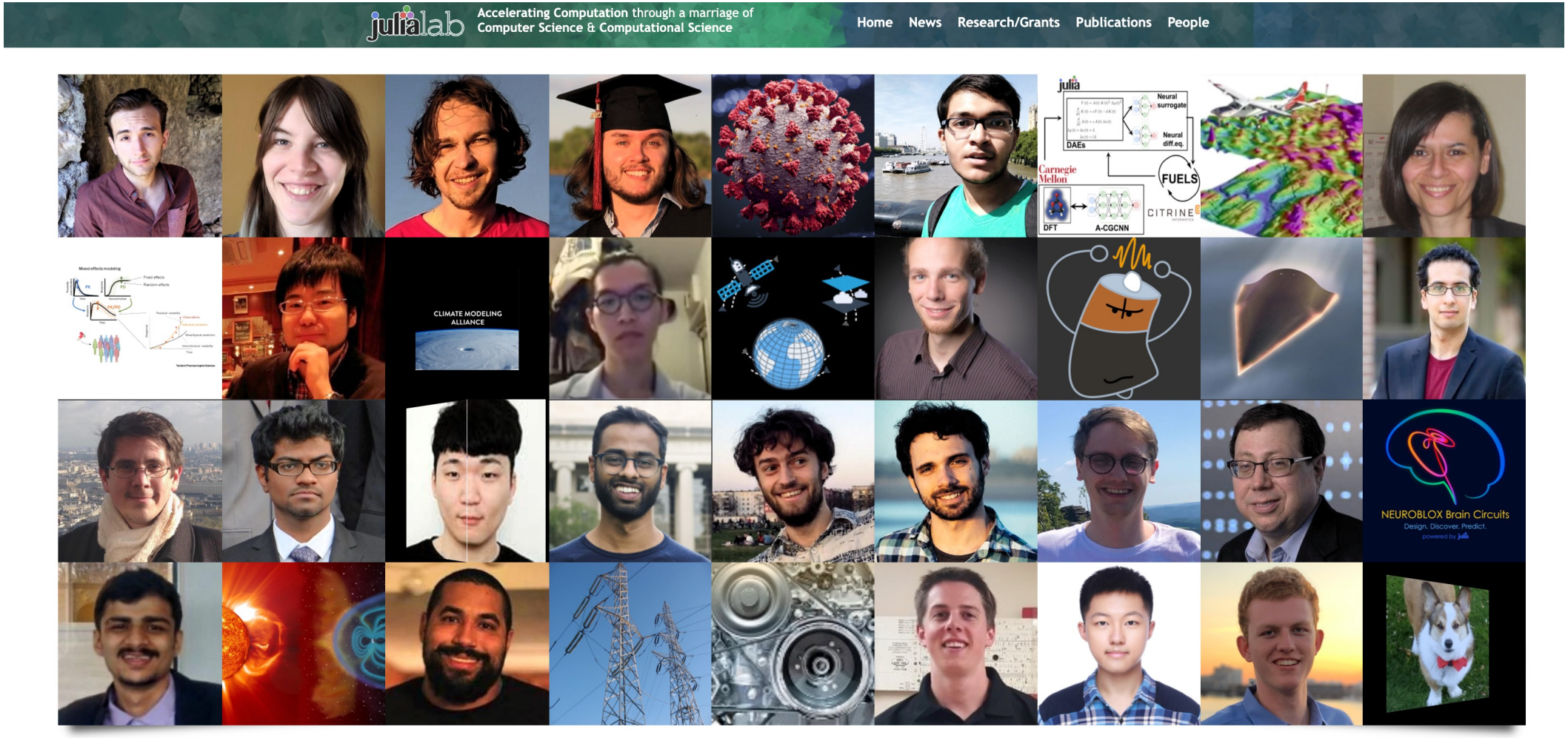
Alan Edelman

SIAM PP 2022

February 23, 2022

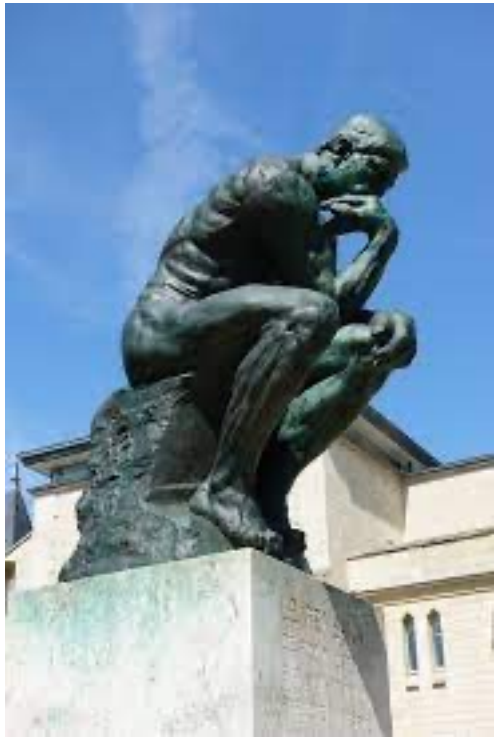
MIT MATHEMATICS

MIT Julialab: julia.mit.edu



Audience Participation

SIAM Conference on Parallel Processing for Scientific Computing (PP22)



Biggest Challenge in our field?

Heterogeneity Some may say …

but what is usually meant by this?

Of course Performance!!!

Heterogeneity Some may say …

but should it be?

Heterogeneity Some may say …

How can we program these complex systems for performance?

Heterogeneity Some may say …

How can we program these complex systems for performance?

Optimizing solely for performance leads to decades of missed opportunities

H. Jackson Brown, Jr

Heterogeneity Some may say …

How can we program these complex systems for performance?

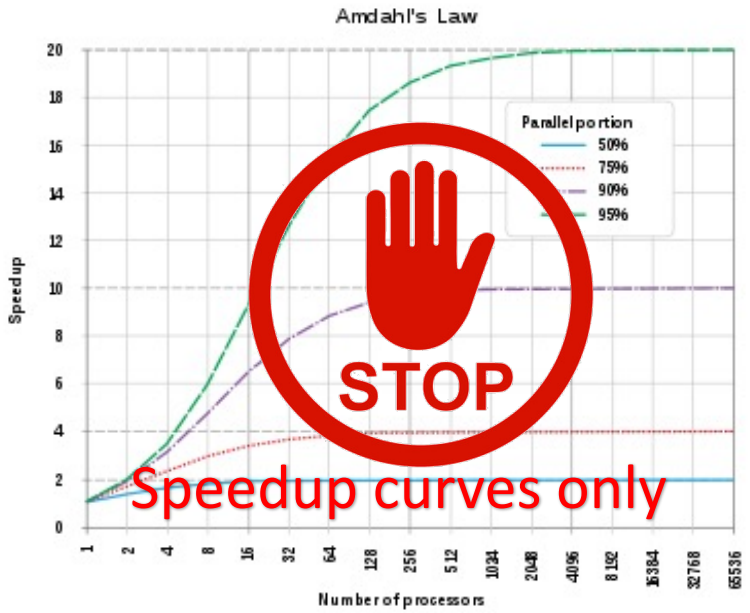
If I could influence the field, I would remove performance as the sole metric of success.

H. Jackson Brown, Jr

Heterogeneity Some may say …

How can we program these complex systems for performance?

If I could influence the field, I would remove performance as the sole metric of success.



Heterogeneity Some may say …

How can we program these complex systems for performance?

I would replace (with discretion) with the size of the user or developer community. (Impact!)
Performance is but one component!

Performance indicator

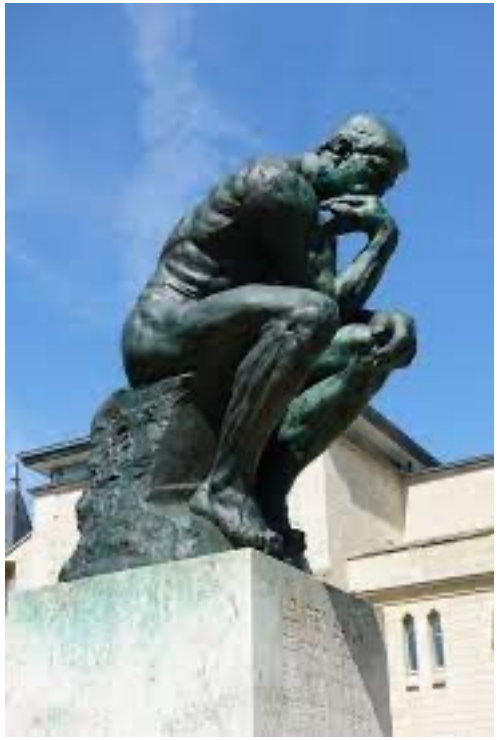
From Wikipedia, the free encyclopedia

Contents [hide]

1 Categorization of indicators
2 Points of measurement
3 Identifying indicators of organization
4 Examples
4.1 Accounts
4.2 Marketing and sales
4.3 Manufacturing
4.4 Professional Services
4.5 System operations
4.6 Project execution
4.7 Supply chain management
4.8 Government
4.9 Further performance indicators

Audience Participation

SIAM Conference on Parallel Processing for Scientific Computing (PP22)



Biggest Challenge in our field?

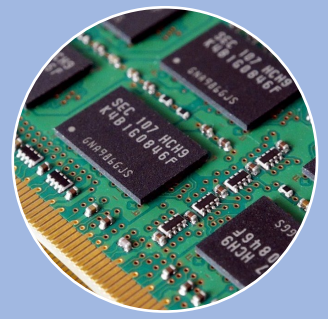
Time for a summary

- Performance alone has lead to opportunity loss
- performance / productivity / reusablility / community /…
- What technically is desirable in the bedrock to achieve the impact we deserve?
- More and more are getting the memo:
- Especially the younger crowd

What about programming models?

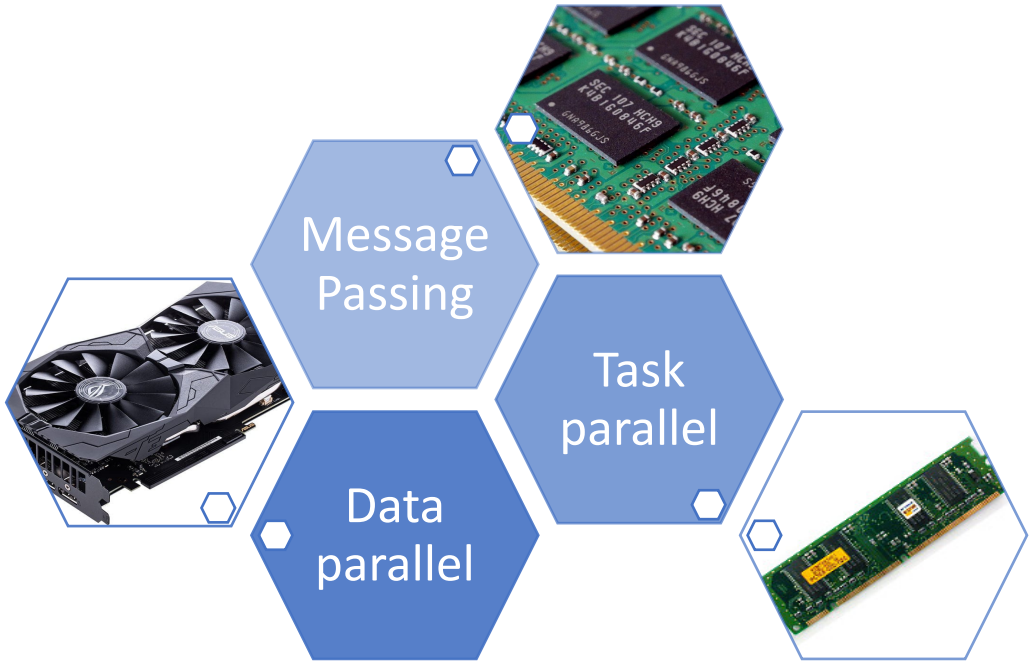
Data parallel – often associated with **GPU**

Task parallel – often associated with **Shared Mem**



Message Passing – often associated with **Distributed Mem**

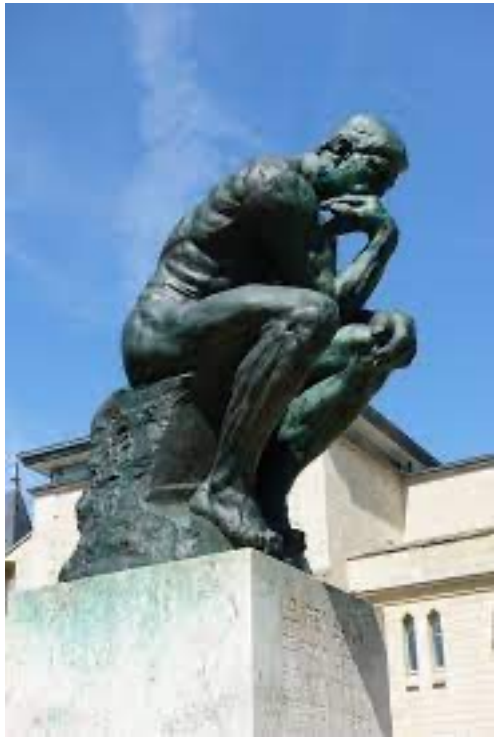
What about programming models?



- Each has their own style
- Difficult to composite due to different styles
- Lots of inconsistencies
- Fragmented communities?
- Difficult to reuse and extend existing work

Audience Participation

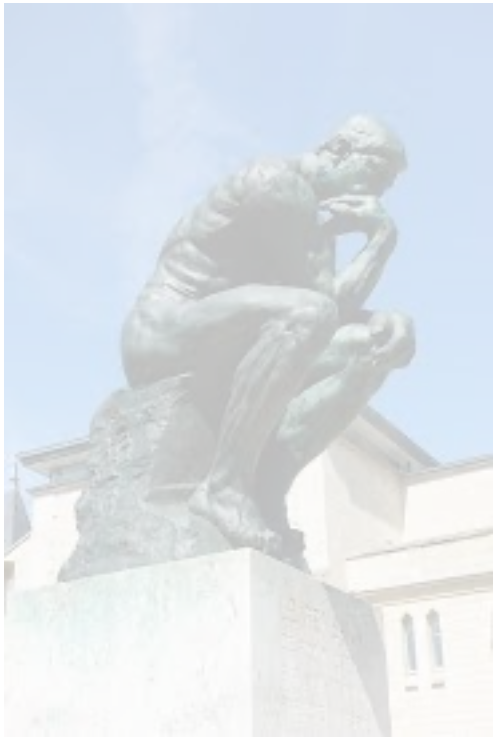
SIAM Conference on Parallel Processing for Scientific Computing (PP22)



Biggest Challenge in our field?

Audience Participation

SIAM Conference on Parallel Processing for Scientific Computing (PP22)



Language!

Biggest Challenge in our field?


productivity
composability
maintainability
performance
community
innovation
Language!

1988 Programming Languages (x2005) 2010 2019

The dark decade

Something seems to have gone wrong!

What did we miss 20 years ago?

Chapel, Fortress and X10:
novel languages for HPC

1 Introduction

Michèle Weiland

In 2002, DARPA[1] launched their High Productivity Computing Systems (HPCS) programme, offering funding for industry and academia to research the development of computing systems which focus on high productivity along with high performance. Part of this programme concentrates on the specification of novel languages for the HPC community. In Phase 2 of the programme (July 2003 - July 2006) the three remaining partners that were awarded funding were Cray, IBM and SUN[2]; SUN were eventually dropped from the programme at the start of Phase 3.[1]

This report will introduce the novel programming languages *Chapel* (Cray), *Fortress* (SUN) and *X10* (IBM) that were developed as part of the HPCS programme.

10 years ago

Why We Created Julia

14 February 2012 | **Jeff Bezanson Stefan Karpinski Viral B. Shah Alan Edelman**

Jeff Bezanson Stefan Karpinski Viral B. Shah Alan Edelman
In short, because we are greedy.

We are power Matlab users. Some of us are Lisp hackers. Some are Pythonistas, others Rubyists, still others Perl hackers. There are those of us who used Mathematica before we could grow facial hair. There are those who still can't grow facial hair. We've generated more R plots than any sane person should. C is our desert island programming language.

We love all of these languages; they are wonderful and powerful. For the work we do — scientific computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing — each one is perfect for some aspects of the work and terrible for others. Each one is a trade-off.

We are greedy: we want more.

We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

(Did we mention it should be as fast as C?)

To solve the parallel computing problem – and it wouldn't be easy.

We'd need help from **the community**.

Today

Keno Fischer (@keno)

Ten years ago, when "Why we created Julia" was published, I was just finishing an exchange student at a tiny school on the Eastern Shore of Maryland. With sc attempting school projects and online courses in Octave, college acceptance in resonated strongly with the need for better tools in computational science and very day. 10 years, there has been hardly a day in the past decade that I did not worked on everything from astronomy, over homomorphic encryption, to simu in between. There have been many memorable events along the way. For exam dynamic language, we pu than a million simultanec to JuliaCon to get togethe tiny group of people hack getting more exciting.

Jose Storopoli (@storopoli)

I stumbled into Julia while trying to do a crazy data transformation in R that was taking forever. I was instantaneously hooked! The syntax is so easy, and it has also a "math" feel to it. You can use all your ϵ and δ, along with the statistician's favorites α and β. Since then, I've become a contributor to the `JuliaStats` and `Turing` (Bayesian modeling) Julia's ecosystems. The community is great and very welcoming. I've made amazing friendships here, co-authored a free open access and open source Julia Data Science book with Rik Huijzer and Lazaro Alonso. The book has been translated to Portuguese and Chinese by volunteers, that which I am grateful and humbled. Julia is taking big strides now, since its first appearance 10 years ago. I am really excited about its future and honored to be a part of it.

Avik Sengupta (@aviks)

My first reaction on seeing the headline on Hacker News that day was a decidedly lukewarm "oh, do we really need yet another programming language". Reading the blog post however piqued my interest – "it surely can't deliver on all that it promises, can it?". It was a slow day at work, so I downloaded the source, and was surprised to see that it built successfully on the first try. My first PR came two weeks later. In adding a new `omputing was certainly possible. In the , as well as being able to write Julia `on it's a

Cristóvão Sousa (@cdsousa)

Julia is the language I was longing for. And in the late summer of 2013, that "Why We Created Julia" was mind-blowing to me! I had been taught to program in C and C++ and later I learned some MATLAB too. I always looked for performance and I'm the kind of person that prefers to waste more time optimizing code than wasting little time waiting for it to run. But on the other hand, I need to write math/scientific code and I like to do it in a high-level language. When starting the PhD in 2009, along with C++ I wanted to use an open-source higher-level language, so I opted for Python. However, the slowness was a bummer and I had to rely on workarounds like Cython. I was not happy, and I was greedy, always wondering "Why can't a better language, high-level but perforn summer of 2013, its d exactly "Oh no! Yet ar and "WOW! YAY!". It gone by, I'm a very ha from the excellent cor

Jerry Ling (@Moelf)

I was, like many people, moderately skeptical when I saw Julia's promise of "the best of both worlds" the first time; in fact, I remember walking away not particularly impressed the first time I tried Julia in 2018, largely due to latency (TTFT) issue with plotting. That soon change when I tried again near v1.0 release after hearing a PhD student (Katharine Hyatt) in UCSB physics was doing cool quantum stuff in Julia. Things moved fast after that: once you know the two-language problem, it's everywhere in physics. So many things are "free" in Julia, 'Python can be skipped because Julia is not slow, auto diff would have helped with `blem in physics etc. Today, though not as proficient in compiler or typing system than s post, I strive to bring better computing tools to physics, and I hope to witness a rapid `e next decade.

Chris Rackauckas (@chrisrackauckas)

I came rather late to the scene, around 2016. I had written GUIs in R, libraries in Python and MATLAB, handled MEX files and wrote my MPI using C. I wrote some ODE solvers using Fortran like a good ol' kid. I did my due diligence. It was an absolute mess, especially being a Windows user, and so when I found Julia I was astonished to find something that would actually work. No more SciPy installation error from some compiler missing on Windows: Julia was the compiler. And more and more, Julia is the compiler. I everything was new. In those early days of v0.5-beta I remember creating the Plots.jl and documentation as trades for getting bug fixes. And from all of that experience in the "wild with language and computation had changed. Before, I was thinking language was fixed a equation solvers (DifferentialEquations.jl) that were "my" work, but with Julia everything implementation of generating random Poisson numbers fast enough, or for this use case faster that hits the statistical tolerance that I needed? Well that code is in Julia, so let's ta everything in one language soon meant the world expanded to thinking about how every operation could be optimized for the application at hand. Everything that was taken for g hands of the domain. That was how differential equation solving expanded into scientific the SciML Open Source Software Organization). Now we have SciML benchmarks along t bottom, incorporating everything from physics-informed neural network solvers of partia equations to recursive array data structures. There are so many places in math that are si because they sound like the domain of a compilers instead of "computational science", an how the next 10 years bridges this gap in Julia.

Julian Samaroo (@jpsamaroo)

About 8 years ago, I was a student studying neuroscience in college, very interested in computational modeling. I had implemented some basic spiking neural network simulators in MATLAB, and ported them to JavaScript, but always found both languages lacking. Still, I persisted with JavaScript, until about 3 years later, when I first heard about Julia from somewhere on the interwebs. Out of boredom and a mild displeasure with JavaScript, I decided to give Julia a try. I will be honest; I had no shortage of problems using it! The errors were unintuitive, the syntax was strange, and it was *slow* (to compile, but what was the difference?). I kept using JavaScript for work, but slowly on the side I started getting more familiar with Julia, and tried using it for some of my simulations and side projects. Skipping to about 4 years ago, I got "nerd-sniped" by a certain Valentin Churavy to implement support for AMD GPUs in Julia, since I was so vocal about wanting to support a more open-source friendly GPU vendor. Over the next 2 years, I developed AMDGPUnative.jl (now AMDGPU.jl) and Dagger.jl in my free time, slowly gaining an appreciation for Julia, and attended my very first JuliaCon in 2019! Somewhat shortly afterwards, Valentin reached out from the JuliaLab and offered me an RSE position (which was an offer I could have only dreamed of at that point in life!), which I gladly accepted. Since then, I've been working full-time on AMDGPU.jl, Dagger.jl, BPFnative.jl, and Julia itself, and have been loving every moment of it. I don't think I could ever give up the wonderful ecosystem, language, and people that I've come to know and love over the last 5 years, and I look forward to many more wonderful years to come.

Matt Brzezinski (@n

July 2019 I left my job at AWS and joinec I first heard about the new programming over their entire code base from Matlab t Julia in a production environment. I spent my first week at Invenia porting various internal and external packages from Julia 0.7 to 1.0 and found it to be a simple and intuitive language to use coming from a Python background. My second week was spent in Baltimore meeting and chatting and learning with the community in Baltimore for JuliaCon 2019. I was hooked. Almost 3 years later I've found that Julia has been my go to language for nearly everything. The ease of use, speed and community keep bringing me back for more. I'm excited to see where it goes from here.

To solve the parallel computing problem – and it wouldn't be easy.

We'd need help from **the community**.

Why We Use Julia, 10 Years Later

14 February 2022 | **The Julia Community**

What did we miss 20 years ago?

A multidisciplinary user base for high productivity & high performance

What did we miss 20 years ago?

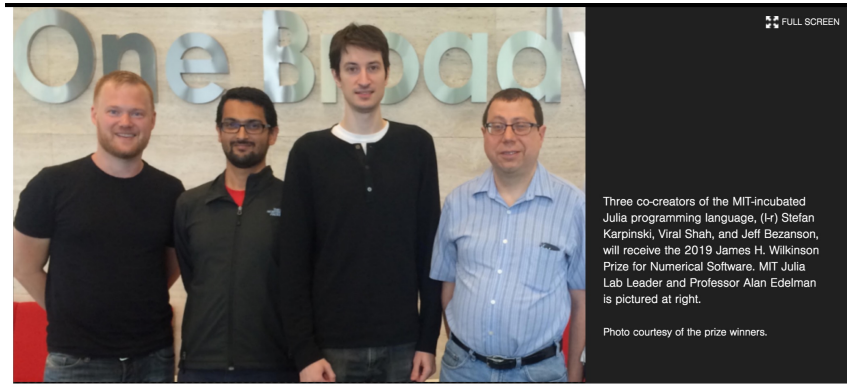
A multidisciplinary user base for high productivity & high performance

multiple dispatch
static + dynamic
type specialization
…

What did we miss 20 years ago?

A multidisciplinary user base for high productivity & high performance

multiple dispatch
static + dynamic
type specialization
…

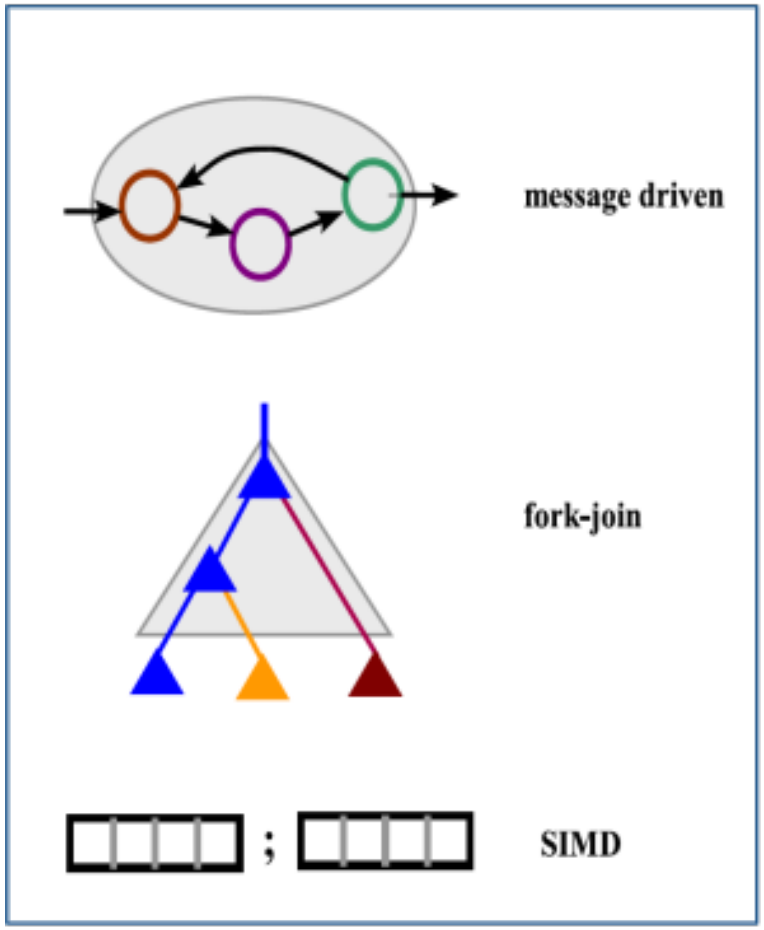


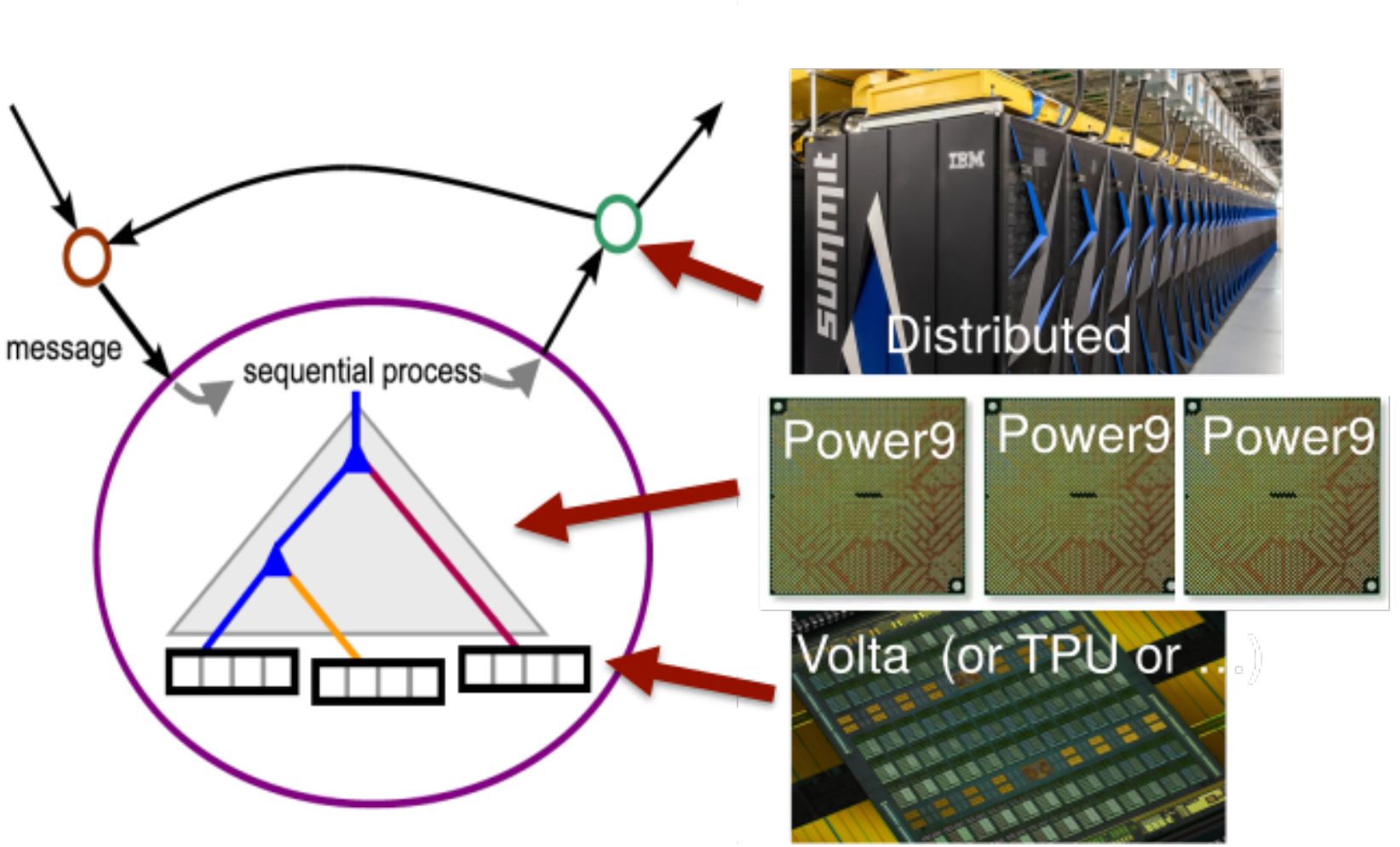
Julia language co-creators win James H. Wilkinson Prize for Numerical Software

"Patience is a virtue."

We knew the right user base starts serial and takes over 10 years to mature.

Back to Heterogeneity: Towards Higher Level Abstractions





Julia HPC

Accelerators

AMDGPU.jl

CUDA.jl

OneAPI.jl

JuliaGPU/
GPUArrays.jl

Reusable array functionality for Julia's various GPU backends.

https://github.com › JuliaGPU › KernelAbstractions

KernelAbstractions.jl - Heterogeneous programming in Julia

Heterogeneous programming in **Julia**. Contribute to JuliaGPU/KernelAbstractions.jl development by creating an ... **JuliaGPU** / **KernelAbstractions**.jl Public.

Shared Mem

JuliaFolds/**FLoops.jl**

Fast sequential, threaded, and distributed for-loops for Julia—fold for humans™

Announcing composable multi-threaded parallelism in Julia

23 July 2019 | **Jeff Bezanson (Julia Computing), Jameson Nash (Julia Computing), Kiran Pamnany (Intel)**

Base / Multi-Threading

Multi-Threading

`Base.Threads.@threads` — Macro

Julia Atomics Manifesto

This proposal aims to define the memory model of Julia and to provide certain guarantees in the presence of data races, both by default and through providing intrinsics to allow the user to specify the level of guarantees required. This should allow native implementation in Julia of simple system primitives (like mutexes), interoperate with native system code, and aim to give generally explainable behaviors without incurring significant performance cost. Additionally, it strives to be general-purpose and yet clear about the user's intent—particularly with respect to ensuring that an atomic-type field is accessed with proper care for synchronization.

Distributed

GitHub - eth-cscs/ImplicitGlobalGrid.jl ...
github.com

JuliaParallel/**MPI.jl**

MPI wrappers for Julia

JuliaParallel/
Dagger.jl

A framework for out-of-core and parallel execution

Standard Library / Distributed Computing

Distributed Computing

Julia HPC

Phase 1: Building up these tools
Phase 2: Letting these tools break out from these separate categories

Accelerators

AMDGPU.jl

CUDA.jl

OneAPI.jl

JuliaGPU/
GPUArrays.jl

Reusable array functionality for Julia's various GPU backends.

https://github.com › JuliaGPU › KernelAbstractions

KernelAbstractions.jl - Heterogeneous programming in Julia

Heterogeneous programming in **Julia**. Contribute to JuliaGPU/KernelAbstractions.jl development by creating an ... **JuliaGPU** / **KernelAbstractions**.jl Public.

Shared Mem

JuliaFolds/**FLoops.jl**

Fast sequential, threaded, and distributed for-loops for Julia—fold for humans™

Announcing composable multi-threaded parallelism in Julia

23 July 2019 | **Jeff Bezanson (Julia Computing), Jameson Nash (Julia Computing), Kiran Pamnany (Intel)**

Base / Multi-Threading

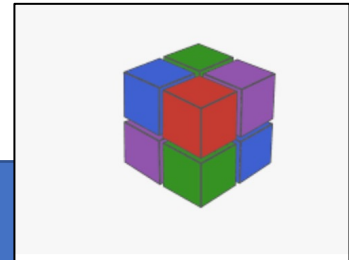
Multi-Threading

`Base.Threads.@threads` — Macro

Julia Atomics Manifesto

This proposal aims to define the memory model of Julia and to provide certain guarantees in the presence of data races, both by default and through providing intrinsics to allow the user to specify the level of guarantees required. This should allow native implementation in Julia of simple system primitives (like mutexes), interoperate with native system code, and aim to give generally explainable behaviors without incurring significant performance cost. Additionally, it strives to be general-purpose and yet clear about the user's intent—particularly with respect to ensuring that an atomic-type field is accessed with proper care for synchronization.

Distributed



GitHub - eth-cscs/ImplicitGlobalGrid.jl ...
github.com

JuliaParallel/**MPI.jl**

MPI wrappers for Julia

JuliaParallel/
Dagger.jl

A framework for out-of-core and parallel execution

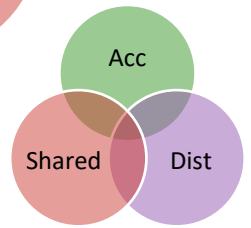
Standard Library / Distributed Computing

Distributed Computing

Julia HPC

Phase 1: Building up these tools

Phase 2: Letting these tools break out from these separate categories



Phase 3: Higher level abstractions

Accelerators

AMDGPU.jl

CUDA.jl

OneAPI.jl

JuliaGPU/
GPUArrays.jl

Reusable array functionality for Julia's various GPU backends.

https://github.com › JuliaGPU › KernelAbstractions

KernelAbstractions.jl - Heterogeneous programming in Julia

Heterogeneous programming in **Julia**. Contribute to JuliaGPU/KernelAbstractions.jl development by creating an ... **JuliaGPU** / **KernelAbstractions**.jl Public.

Shared Mem

JuliaFolds/**FLoops.jl**

Fast sequential, threaded, and distributed for-loops for Julia—fold for humans™

Announcing composable multi-threaded parallelism in Julia

23 July 2019 | **Jeff Bezanson (Julia Computing), Jameson Nash (Julia Computing), Kiran Pamnany (Intel)**

Base / Multi-Threading

Multi-Threading

`Base.Threads.@threads` — Macro

Julia Atomics Manifesto

This proposal aims to define the memory model of Julia and to provide certain guarantees in the presence of data races, both by default and through providing intrinsics to allow the user to specify the level of guarantees required. This should allow native implementation in Julia of simple system primitives (like mutexes), interoperate with native system code, and aim to give generally explainable behaviors without incurring significant performance cost. Additionally, it strives to be general-purpose and yet clear about the user's intent—particularly with respect to ensuring that an atomic-type field is accessed with proper care for synchronization.

Distributed

GitHub - eth-cscs/ImplicitGlobalGrid.jl ...
github.com

JuliaParallel/**MPI.jl**

MPI wrappers for Julia

JuliaParallel/
Dagger.jl

A framework for out-of-core and parallel execution

Standard Library / Distributed Computing

Distributed Computing

Case study: Histogram diag(qr(randn(n,n)))

can't help myself, I ❤ randomized linear algebra

$$H_1 H_2 \cdots H_{n-1} H_n \times \texttt{randn(n,n)} = R_n \sim \begin{pmatrix} \chi_n & \mathrm{G} & \mathrm{G} & \cdots & \mathrm{G} & \mathrm{G} & \mathrm{G} \\ & \chi_{n-1} & \mathrm{G} & \cdots & \mathrm{G} & \mathrm{G} & \mathrm{G} \\ & & \chi_{n-2} & \cdots & \mathrm{G} & \mathrm{G} & \mathrm{G} \\ & & & \ddots & \vdots & \vdots & \vdots \\ & & & & \chi_3 & \mathrm{G} & \mathrm{G} \\ & & & & & \chi_2 & \mathrm{G} \\ & & & & & & \chi_1 \end{pmatrix}$$

H: Householder reflector

G: standard normal

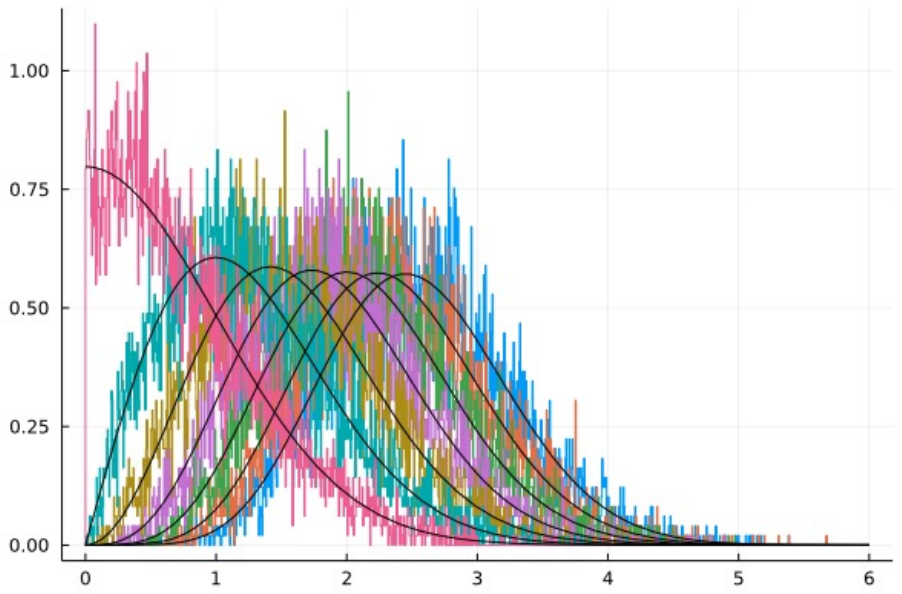
Write Once, Execute Anywhere!

```
function qrhist(;
    nsamples = 2^13,
    nbins = 1_000,
    xmin::Real = 0.0,
    xmax::Real = 6.0,
    matsize::Val{m} = Val(7),
    executor = nothing,
) where {m}
    hist = fill!(arrayon(executor, Int, (nbins, m)), 0)
    dx = (xmax - xmin) / nbins
    @floop executor for _i in 1:nsamples
        M = randn(SMatrix{m,m,Float32})
        _q, r = qr(M)
        for i in 1:m
            bin = floor(Int, (abs(r[i, i]) - xmin) / dx) + 1
            @inbounds @atomic :monotonic hist[max(begin, min(bin, end)), i] += 1
        end
    end
    edges = collect(xmin:dx:xmax)
    return [Histogram(edges, collect(h), :left, false) for h in eachcol(hist)]
end
```

Execution on a single CPU

```
In [3]: hist = normalize.(qrhist(executor = SequentialEx(), nsamples = 2^13))

        plt = plot(hist, seriestype = :step, legend = nothing)
        plot_chi!(plt, length(hist))
```

Out[3]:



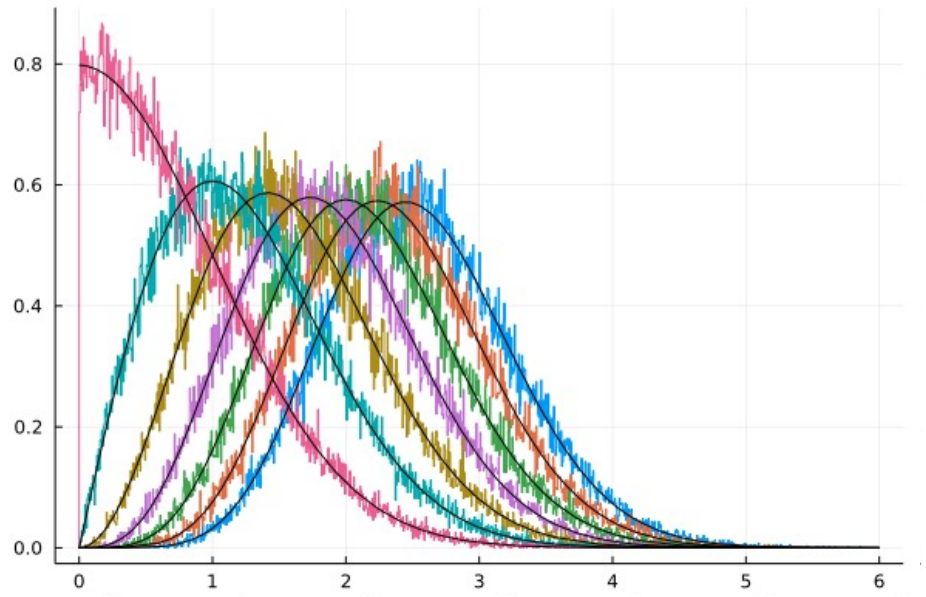
Write Once, Execute Anywhere!

```
function qrhist(;
    nsamples = 2^16,
    nbins = 1_000,
    xmin::Real = 0.0,
    xmax::Real = 6.0,
    matsize::Val{m} = Val(7),
    executor = nothing,
) where {m}
    hist = fill!(arrayon(executor, Int, (nbins, m)), 0)
    dx = (xmax - xmin) / nbins
    @floop executor for _i in 1:nsamples
        M = randn(SMatrix{m,m,Float32})
        _q, r = qr(M)
        for i in 1:m
            bin = floor(Int, (abs(r[i, i]) - xmin) / dx) + 1
            @inbounds @atomic :monotonic hist[max(begin, min(bin, end)), i] += 1
        end
    end
    edges = collect(xmin:dx:xmax)
    return [Histogram(edges, collect(h), :left, false) for h in eachcol(hist)]
end
```

Execution on multiple CPUs

```
In [4]: hist = normalize.(qrhist(nsamples = 2^16))

        plt = plot(hist, seriestype = :step, legend = nothing)
        plot_chi!(plt, length(hist))
```

Out[4]:



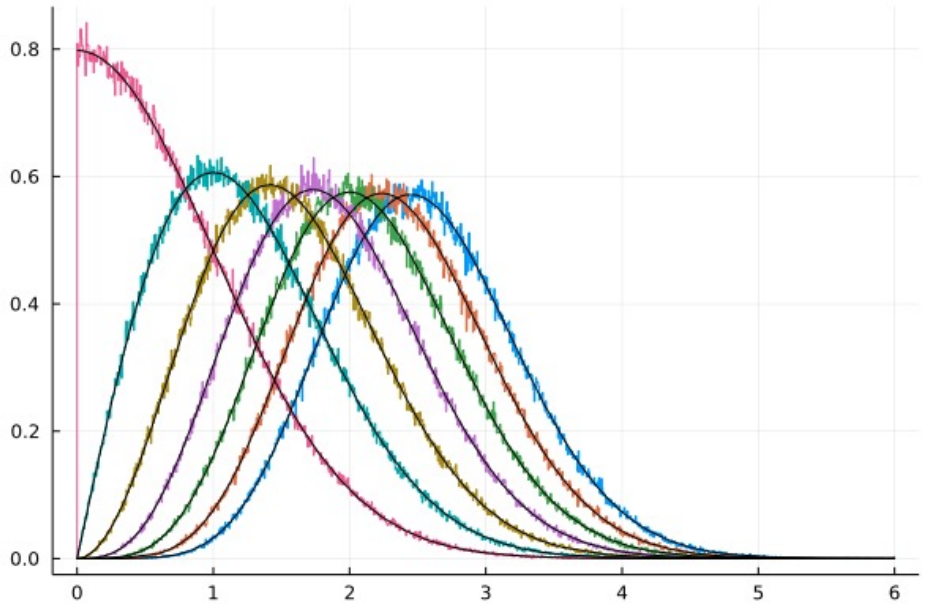
Write Once, Execute Anywhere!

```
function qrhist(;
    nsamples = 2^18,
    nbins = 1_000,
    xmin::Real = 0.0,
    xmax::Real = 6.0,
    matsize::Val{m} = Val(7),
    executor = nothing,
) where {m}
    hist = fill!(arrayon(executor, Int, (nbins, m)), 0)
    dx = (xmax - xmin) / nbins
    @floop executor for _i in 1:nsamples
        M = randn(SMatrix{m,m,Float32})
        _q, r = qr(M)
        for i in 1:m
            bin = floor(Int, (abs(r[i, i]) - xmin) / dx) + 1
            @inbounds @atomic :monotonic hist[max(begin, min(bin, end)), i] += 1
        end
    end
    edges = collect(xmin:dx:xmax)
    return [Histogram(edges, collect(h), :left, false) for h in eachcol(hist)]
end
```

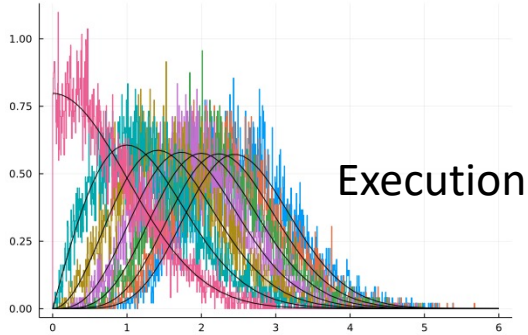
Execution on GPU

```
In [5]: using AtomicArraysCUDA
        using CUDA
        using FoldsCUDA
        arrayon(::CUDAEx, T, size) = CuArray{T}(undef, size)

        hist = normalize.(qrhist(executor = CUDAEx(), nsamples = 2^19))

        plt = plot(hist, seriestype = :step)
        plot_chi!(plt, length(hist))
```

Out[5]:

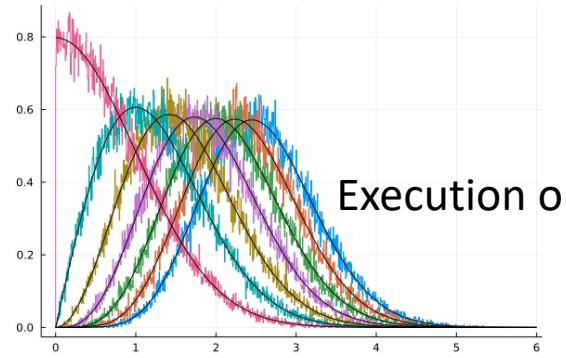


No sacrifice in Performance!



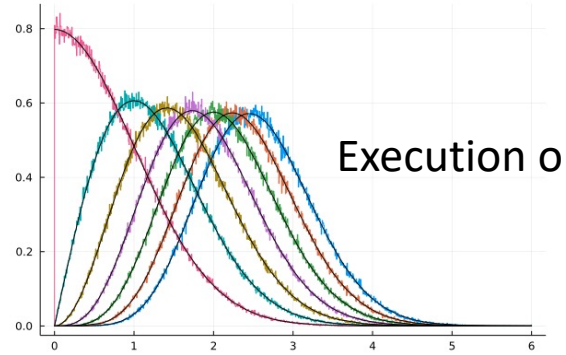
Execution on a single CPU

```
BenchmarkTools.Trial: 550 samples with 1 evaluation.
 Range (min … max):  9.045 ms … 10.760 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     9.082 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   9.090 ms ± 76.303 µs  ┊ GC (mean ± σ):  0.00% ± 0.00%

9.05 ms          Histogram: frequency by time         9.17 ms <

Memory estimate: 118.73 KiB, allocs estimate: 18.
```



Execution on multiple CPUs

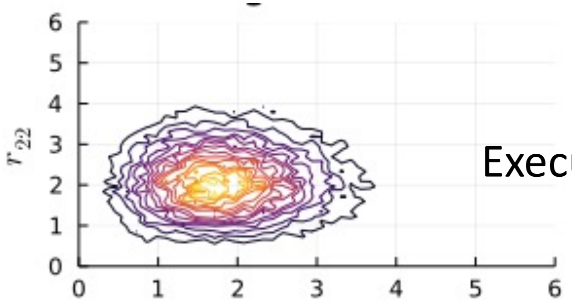
```
BenchmarkTools.Trial: 373 samples with 1 evaluation.
 Range (min … max):  11.491 ms … 59.282 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     12.846 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   13.417 ms ±  3.219 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

11.5 ms          Histogram: log(frequency) by time      25.9 ms <

Memory estimate: 127.33 KiB, allocs estimate: 128.
```



Execution on GPU

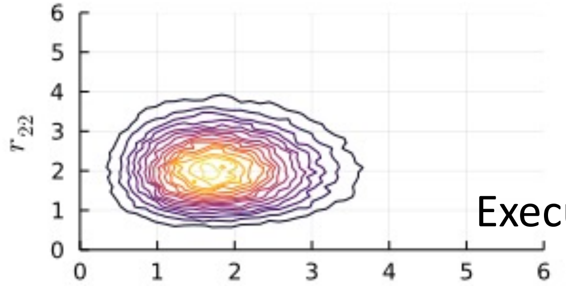
```
BenchmarkTools.Trial: 517 samples with 1 evaluation.
 Range (min … max):  9.202 ms … 32.841 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     9.487 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   9.655 ms ±  1.574 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

9.2 ms           Histogram: log(frequency) by time      16.6 ms <

Memory estimate: 138.11 KiB, allocs estimate: 391.
```

Even more prominent in 2D!



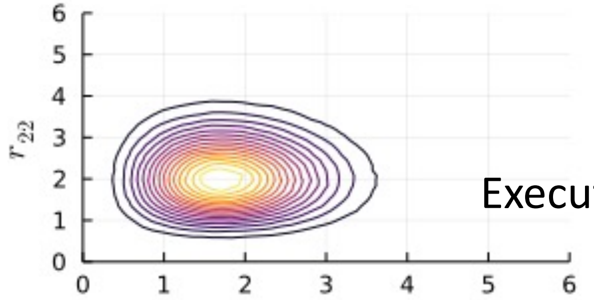
Execution on a single CPU

```
BenchmarkTools.Trial: 213 samples with 1 evaluation.
 Range (min … max):  22.973 ms … 65.061 ms  ┊ GC (min … max): 0.00% … 64.04%
 Time  (median):     23.228 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   23.474 ms ±  2.889 ms  ┊ GC (mean ± σ):  0.83% ±  4.39%

 23 ms        Histogram: log(frequency) by time       26.3 ms <

 Memory estimate: 488.42 KiB, allocs estimate: 3.
```



Execution on multiple CPUs

```
BenchmarkTools.Trial: 164 samples with 1 evaluation.
 Range (min … max):  25.780 ms … 75.986 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     30.574 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   30.560 ms ±  5.035 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

 25.8 ms        Histogram: frequency by time        46.2 ms <

 Memory estimate: 497.02 KiB, allocs estimate: 113.
```



Execution on GPU

```
BenchmarkTools.Trial: 185 samples with 1 evaluation.
 Range (min … max):  26.943 ms … 27.240 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     27.067 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   27.065 ms ± 43.939 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%

 26.9 ms        Histogram: frequency by time        27.2 ms <

 Memory estimate: 19.84 KiB, allocs estimate: 354.
```

Productivity: Focus not on speedup but what can be accomplished / unit time

Showing Impact of Language

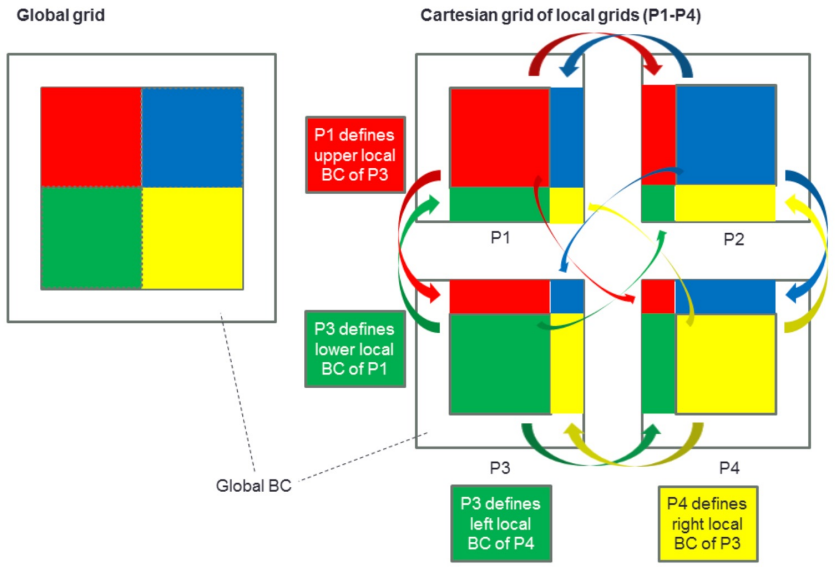
- Case Study: A Random matrix histogram
  - No extra hoops to jump through for parallelism and GPU acceleration (no CUDA, no MPI,…)
  - distribution of diagonals of qr(randn(n,n))

Showing Impact of Language

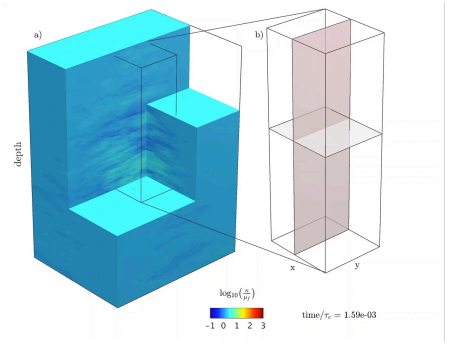
- Case Study: A Random matrix histogram
  - No extra hoops to jump through for parallelism and GPU acceleration (no CUDA, no MPI,…)
  - distribution of diagonals of qr(randn(n,n))
- Demo: Diffusion
  - No Reinventing the Wheel – Build upon solid foundation
    - You have a C/MPI or Fortran/MPI Code? No need to rewrite
    - You need a ghost/halo code? No need to write
  - Can easily compose parallel modes
    - Message Passing and Task Parallelism

ImplicitGlobalGrid

- Samuel Omlin & Ludovic Räss Swiss Computing Center
- The most common HPC request: stencils & halo/ghost cells
- Made nearly trivial!



5



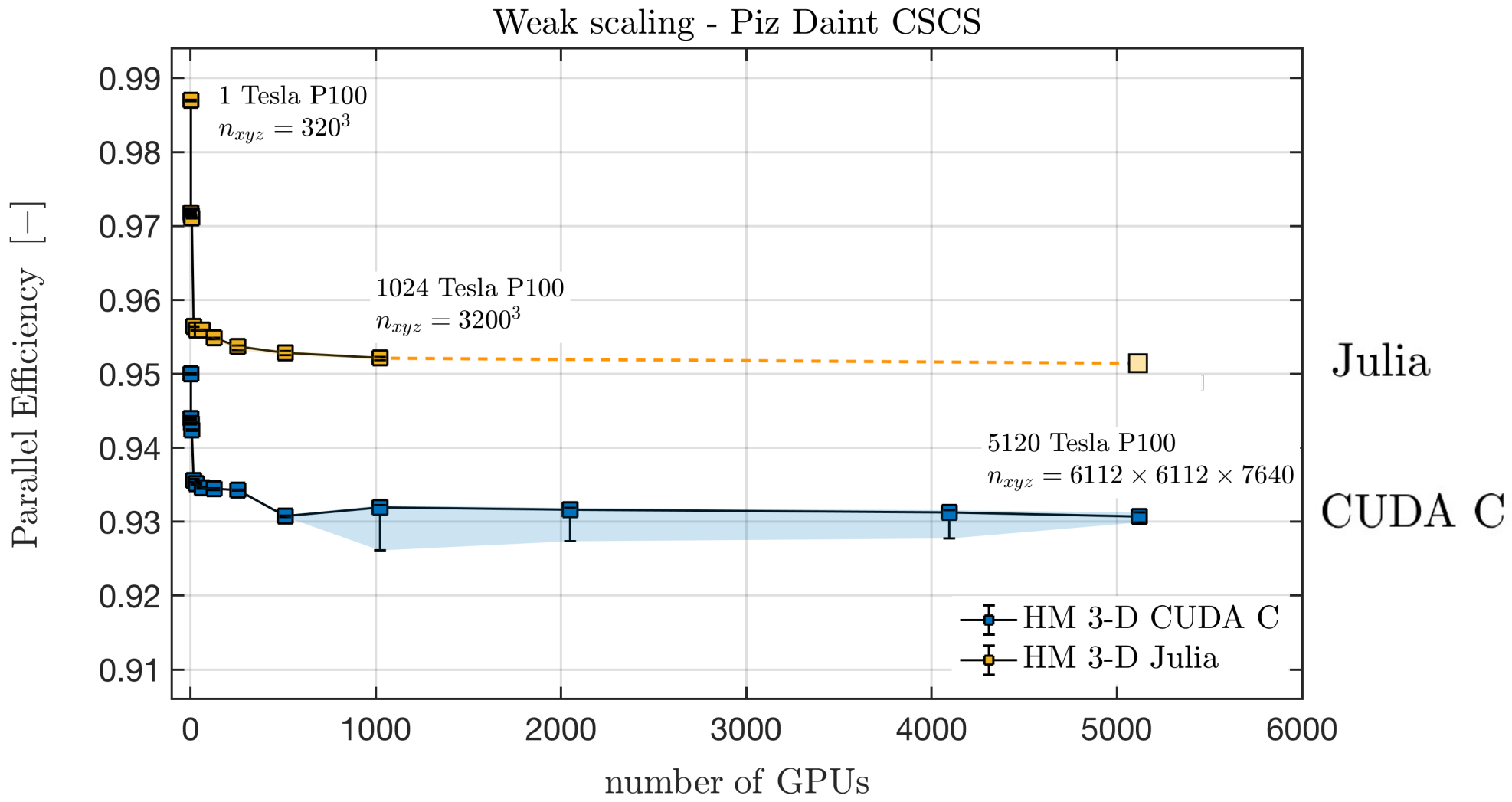
Spontaneous formation of fluid escape pipes from subsurface reservoirs

Ludovic Räss ✉, Nina S. C. Simon & Yury Y. Podladchikov

Hide communication:

- Compute boundary region and interior points on two different streams.
- As soon as boundary region has finished, start halo update

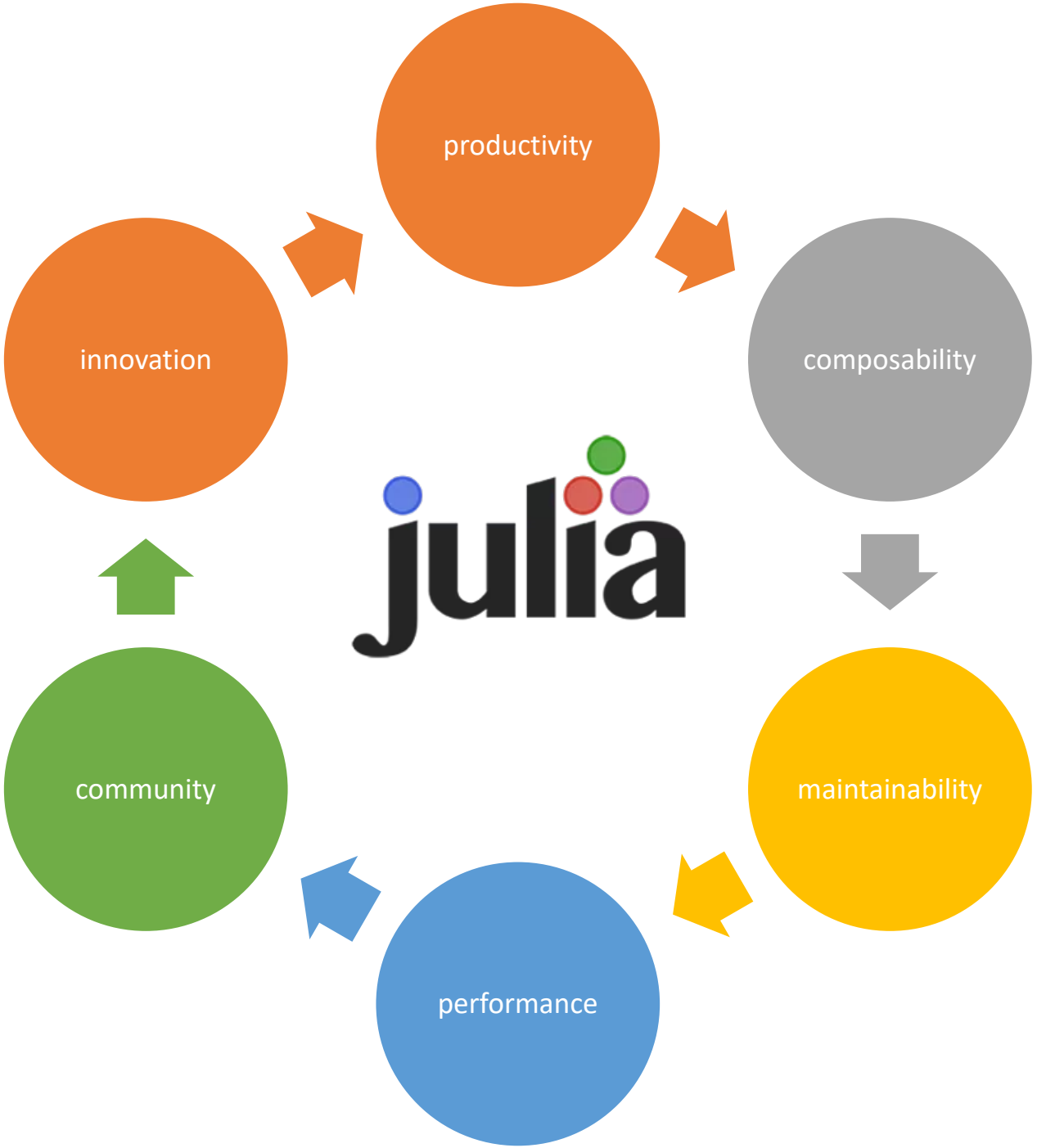
Reported Results
(ImplicitGlobalGrid.jl github)



Demos: Take home message

- You have a C/MPI or Fortran/MPI? No need to rewrite!
- You need a ghost / halo code? No need to rewrite!
- You want to compose different parallel modes? Easy!

No re-inventing the wheel – Build upon solid foundations



Solving Big Challenges in our times

Building and deploying a scalable, composable workflow for CESMIX in Julia

How do we allocate resources and schedule subsystems?

How do we make reproducibility simple?

How do we support cross-language AD?

How do we not accidentially DDOS a cluster?

How do provide binaries for all platforms?

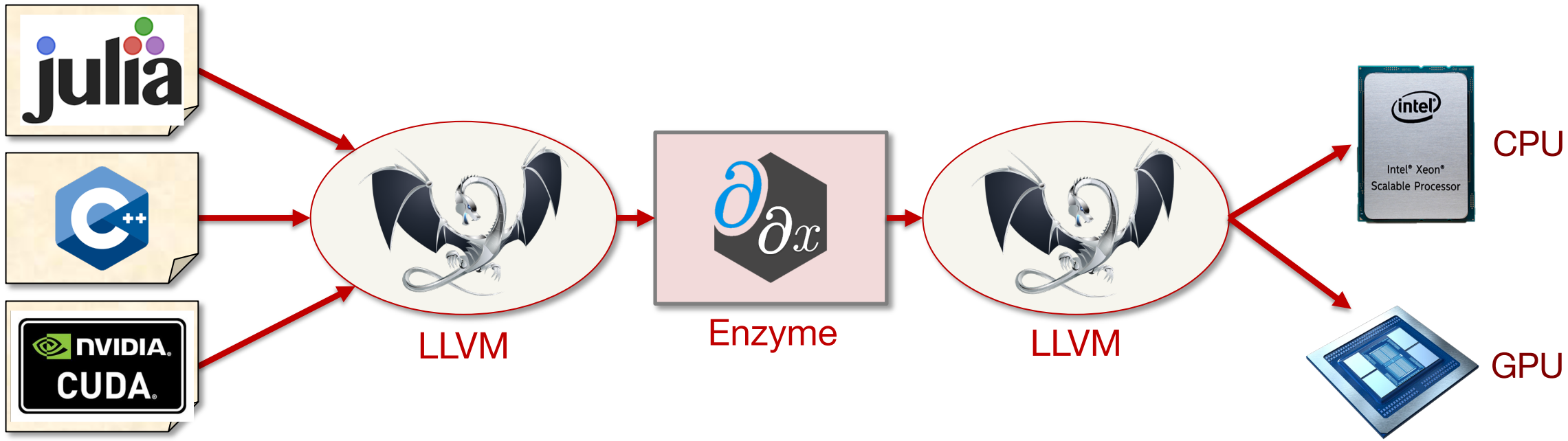
How do we empower user?

Valentin Churavy
JuliaLab, CSAIL
vchuravy@mit.edu

October 2021

Computing derivatives: Enzyme [MC20]

Enzyme performs automatic differentiation (AD) within the compiler.



Enzyme differentiates through MPI.

[MC20] Moses, Churavy. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. NeurIPS, 2020.

MIT News: New climate model to be built from the ground up

MIT News
ON CAMPUS AND AROUND THE WORLD

December 12, 2018

To take advantage of new computer architectures, languages, and machine learning techniques, the team has partnered with Alan Edelman's group in CSAIL at MIT, who will help write the new generation climate model in the Julia computing language developed by the group. This will enable the MIT team to target GPUs, CPUs, and evolving computer architectures within one code base.

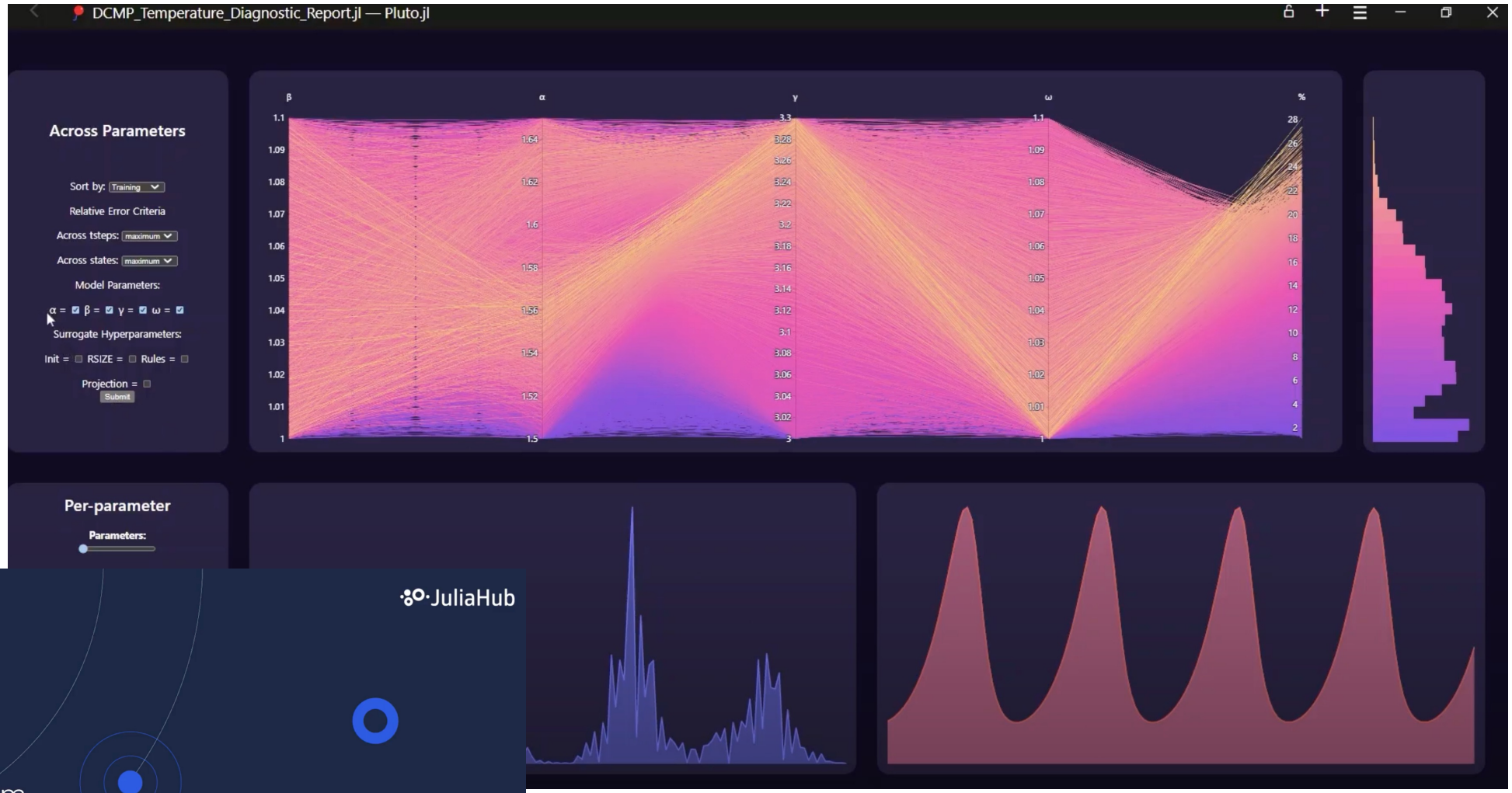

Tapio Schneider

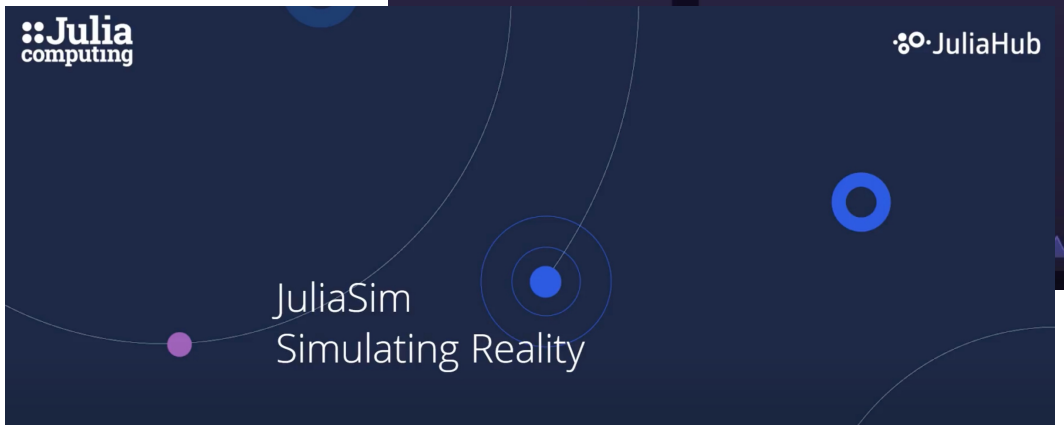

Chris Hill


Raffaele Ferrari

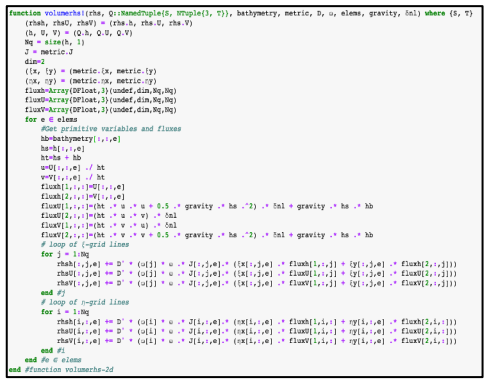

John Marshall

JuliaSim Simulating Reality





Code is the “New” Math



```
Δh[elem]  += ∫∇Ψ(dX⃗ * Ū⃗ₑ * J)
ΔŪ[elem]  += ∫∇Ψ(dX⃗ * (Ū⃗ₑ * Ū⃗ₑ' / htₑ + g * (htₑ^2 - hbₑ^2)/2 * I) * J)
```

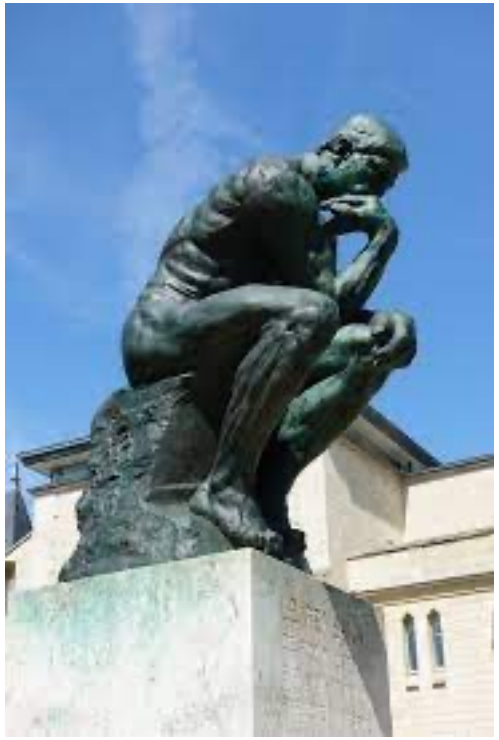
$$\int_{\Omega_e} \nabla\psi \cdot \mathbf{f}_N^{(e)} d\Omega_e$$

$$\mathbf{f} = \left(\mathbf{U}, \frac{\mathbf{U} \otimes \mathbf{U}}{h} + g(h^2 - h_b^2)\mathbf{I}_2 \right)$$

You don’t “code” the “math,” “code” is the “math”

Conclusion

SIAM Conference on Parallel Processing for Scientific Computing (PP22)

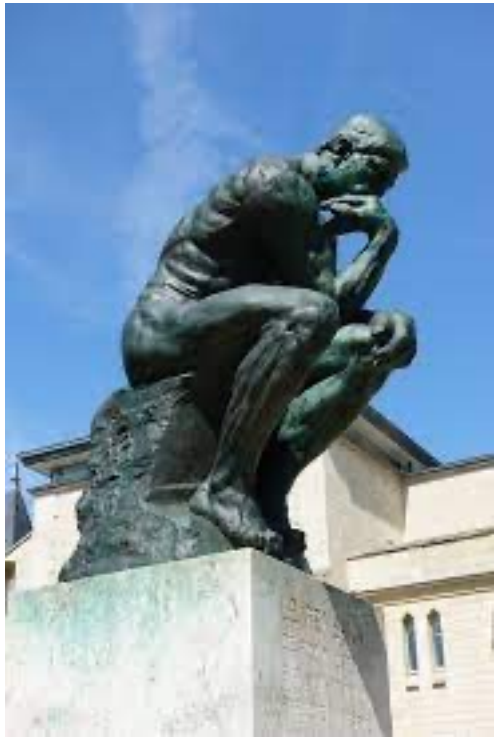


Biggest Challenge in our field?

Making Parallel Computing Easier

Conclusion

SIAM Conference on Parallel Processing for Scientific Computing (PP22)



Biggest Challenge in our field?

Making Parallel Computing Easier

Join the great community: julia