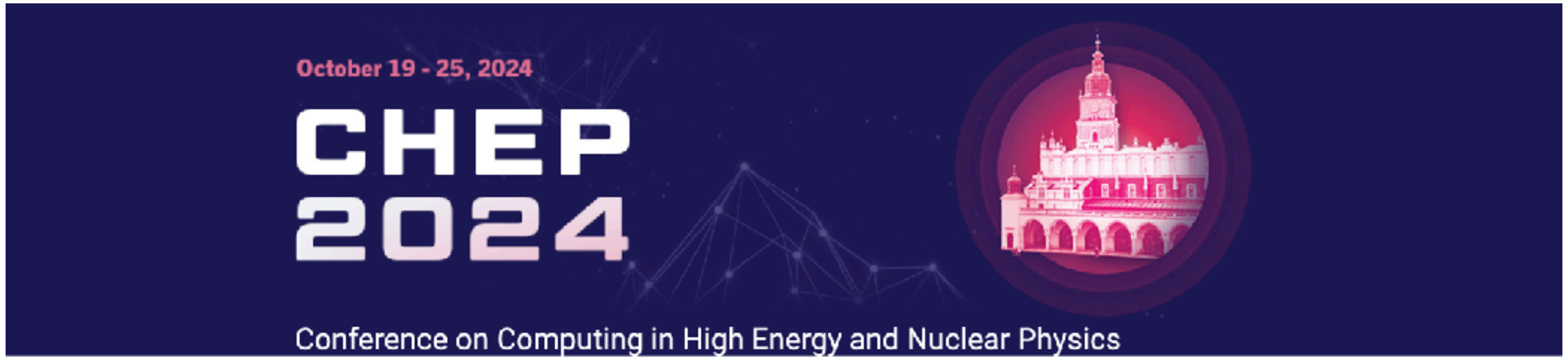
Navigating the Multilingual Landscape of Scientific Computing:

Python, Julia, and Awkward Array

Ianna Osborne, Jim Pivarski, Jerry 🦑 Ling



Julia at CHEP2024 and CHEP2023

a new trend?

- Julia in HEP by Graeme A Stewart, 21 Oct 2024, Plenary session
- R&D towards heterogenous frameworks for Future Experiments by Mateusz Jakub Fila, 21 Oct 2024, Parallel (Track 3)
- ROOT RNTuple implementation in Julia programming language by Jerry Ling, 21 Oct 2024, Parallel (Track 5)
- Comparative efficiency of HEP codes across languages and architectures by Samuel Cadellin Skipsey, 21 Oct 2024, Parallel (Track 6)
- EDM4hep.jl: Analysing EDM4hep files with Julia by Pere Mato, 21 Oct 2024, Poster session
- Fast Jet Reconstruction in Julia by Graeme A Stewart, 23 Oct 2024, Parallel (Track 3)
- BAT.jl, the Bayesian Analysis Toolkit in Julia by Oliver Schulz, 23 Oct 2024, Parallel (Track 5)
- Navigating the Multilingual Landscape of Scientific Computing: Python, Julia, and Awkward Array, by Ianna Osborne, 24 Oct 2024, Parallel (Track 9)

PRINCETON UNIVERSITY

iris hep

Embedding Julia in Python

How easy is it to blend these languages?

- We can use PythonCall for integrating Python's vast ecosystem into Julia projects and JuliaCall for embedding high-performance Julia code into Python scripts.

[ ]:
```
from juliacall import Main as jl
```

[ ]:
```
%load_ext juliacall
```

[ ]:
```
%%julia

using Pkg
Pkg.add("UnROOT")
using UnROOT
```

PRINCETON UNIVERSITY

Using Julia Packages from Python

[1]:
```
from juliacall import Main as jl
```

```
Detected IPython. Loading juliacall extension. See https://juliapy.github.io/PythonCall.jl/stable/compat/#IPython
```

[2]:
```
%load_ext juliacall
```

```
WARNING: replacing module _ipython.
```

[3]:
```
%%julia

using Pkg
Pkg.add("UnROOT")
using UnROOT
```

```
   Resolving package versions...
  No Changes to `~/anaconda3/envs/julia_hep_2024/julia_env/Project.toml`
  No Changes to `~/anaconda3/envs/julia_hep_2024/julia_env/Manifest.toml`
```

PRINCETON UNIVERSITY

ROOT File as Julia Object in Python

Using UnROOT

- This dataset contains about 60 mio. data events from the CMS detector taken in 2012 during Run B and C. The original AOD dataset is converted to the NanoAOD format and reduced to the muon collections.
- Wunsch, Stefan; (2019). DoubleMuParked dataset from 2012 in NanoAOD format reduced on muons. CERN Open Data Portal. DOI:10.7483/OPENDATA.CMS.LVG5.QT81

```
[175]: %%julia

using UnROOT

@time big_tree = ROOTFile("../../../Run2012BC_DoubleMuParked_Muons.root")

  0.007673 seconds (4.65 k allocations: 10.119 MiB)
[175]: ROOTFile with 2 entries and 17 streamers.
../../../Run2012BC_DoubleMuParked_Muons.root
└─ Events (TTree)
   ├─ "nMuon"
   ├─ "Muon_pt"
   ├─ "Muon_eta"
   ├─ "Muon_phi"
   ├─ "Muon_mass"
   └─ "Muon_charge"
```

```
[176]: jl.big_tree
[176]: ROOTFile with 2 entries and 17 streamers.
../../../Run2012BC_DoubleMuParked_Muons.root
└─ Events (TTree)
   ├─ "nMuon"
   ├─ "Muon_pt"
   ├─ "Muon_eta"
   ├─ "Muon_phi"
   ├─ "Muon_mass"
   └─ "Muon_charge"
```

```
[174]: %%julia

@time events = LazyTree(big_tree, "Events")

  0.000334 seconds (365 allocations: 31.703 KiB)
```

[174]: 61,540,413 rows × 6 columns (omitted printing of 61,540,403 rows)

	Muon_phi SubArray{Float3	nMuon UInt32	Muon_pt SubArray{Float3	Muon_eta SubArray{Float3	Muon_charge SubArray{Int32,	Muon_mass SubArray{Float3
1	[-0.0343, 2.54]	2	[10.8, 15.7]	[1.07, -0.564]	[-1, -1]	[0.106, 0.106]
2	[-0.275, 2.54]	2	[10.5, 16.3]	[-0.428, 0.349]	[1, -1]	[0.106, 0.106]
3	[-1.22]	1	[3.28]	[2.21]	[1]	[0.106]
4	[-2.08, 0.251, -2.01, -1.85]	4	[11.4, 17.6, 9.62, 3.5]	[-1.59, -1.75, -1.59, -1.66]	[1, 1, 1, 1]	[0.106, 0.106, 0.106, 0.106]

Julia ROOT Tree in Python

Faster way to read ROOT files

```
[7]: events = jl.Main.LazyTree(file, "Events")

[8]: %%timeit
     jl.Main.LazyTree(file, "Events")

     368 µs ± 23.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- With viewing the data as AwkwardArray we can use either Julia or Python analysis code or even combine both languages.
- Getting the best performance from Julia requires us to focus on type-stability and good practices for reducing unnecessary recompilation.

PRINCETON UNIVERSITY

iris hep

Including Julia Code in Python

Notes on code organization

```julia
# Predefine the output structure with a concrete NamedTuple type
const RecordArrayType = NamedTuple{(:pt, :eta, :phi, :mass, :charge, :isolation)}

function make_record_array(
  events::NamedTuple{(:muon,),
  Tuple{
     NamedTuple{(:pt, :eta, :phi, :mass, :charge, :pfRelIso03_all),
       Tuple{
         Vector{T}, Vector{T}, Vector{T}, Vector{T}, Vector{T}, Vector{T}
       }
     }
   }
 }) where T

    # Convert the relevant fields into AwkwardArray arrays
    array = AwkwardArray.RecordArray(
      RecordArrayType((
        AwkwardArray.from_iter(events.muon.pt),
        AwkwardArray.from_iter(events.muon.eta),
        AwkwardArray.from_iter(events.muon.phi),
        AwkwardArray.from_iter(events.muon.mass),
        AwkwardArray.from_iter(events.muon.charge),
        AwkwardArray.from_iter(events.muon.pfRelIso03_all)
      ))
    )

    return array
end
```

```python
[20]: jl.include('awkward_analyzer_functions.jl');
```

- Provide the correct path when using the include function.
- If your project grows larger, consider structuring your code into more modules and files for better organization:
  - It is generally a good practice to organize your code into modules. This helps with namespace management and reduces the likelihood of name collisions.
  - Use export to expose functions from a module. This makes it easy to access the desired functionality after including a module.

AwkwardArray.jl as Data Bridge

Between Julia and Python

Awkward Array.jl

for Julia!

Search docs

Version dev

Getting started

Edit on GitHub

Getting started

Let's assume that both Python and Julia are installed.

Note

If Julia is not install it is recomended to follow its official installation instructions described here.

Installation

It is recommended to use conda virtul environment.

Using Julia Awkward Arrays from Python

- To install Awkward Array Python package:

```
conda install -c conda-forge awkward
```

- To install JuliaCall:

```
conda install pyjuliacall
```

JuliaCall takes care of installing all necessary Julia packages, including this package.

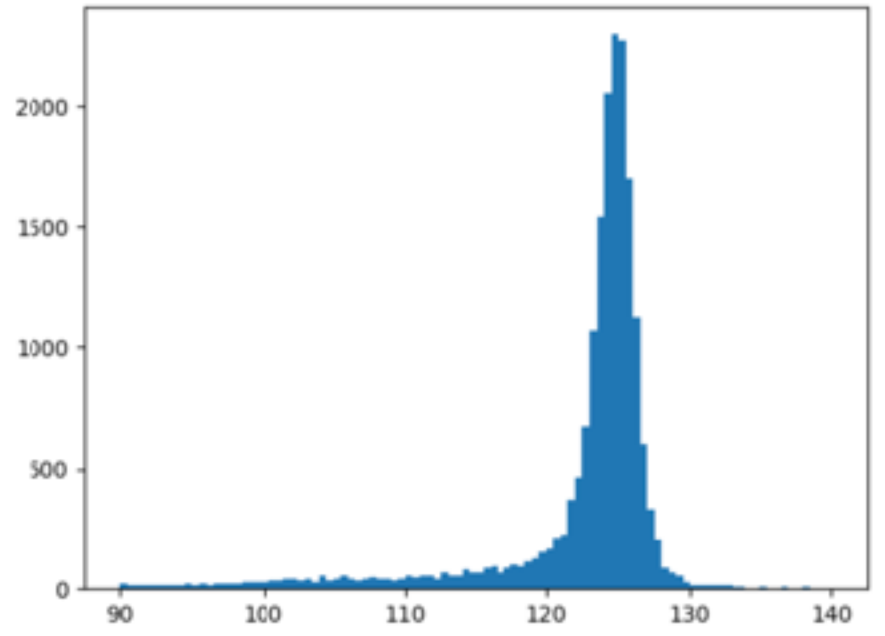
```
import awkward as ak
from juliacall import Main as jl

jl.seval("using AwkwardArray")
```

- Using AwkwardArray in Julia to calculate Higgs mass:

```
[16]: import matplotlib.pyplot as plt
```

```
[17]: %%timeit
plt.hist(jl.array, bins=100, range=(90, 140));
```

66.9 ms ± 8.05 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)




```
function main_looper(events)
    array = AwkwardArray.PrimitiveArray{Float64}()
    for evt in events

        (; Muon_charge) = evt
        if length(Muon_charge) != 4
            continue
        end
        sum(Muon_charge) != 0 && continue # shortcut if-else

        (; Muon_pt, Muon_eta, Muon_phi, Muon_mass) = evt
        higgs_4vector = sum(LorentzVectorCyl.(Muon_pt, Muon_eta, Muon_phi, Muon_mass))
        higgs_mass = mass(higgs_4vector)

        push!(array, higgs_mass)
    end

    return array
end
```

PRINCETON UNIVERSITY

Calling Julia from Python Efficiency

with a very small overhead

	Call Julia from Python	Julia	Julia precompiled
Open ROOT file with UnROOT.ROOTFile	1.05 ms ± 28.5 μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)	0.047574 seconds (6.00 k allocations: 565.602 KiB, 94.32% compilation time)	0.001293 seconds (4.78 k allocations: 509.227 KiB)
Get a tree with UnROOT.LazyTree	368 μs ± 23.2 μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)	0.520122 seconds (468.07 k allocations: 32.848 MiB, 99.26% compilation time)	0.000502 seconds (2.40 k allocations: 230.648 KiB)
Execute Julia function main_looper	CPU times: user 430 ms, sys: 22 ms, total: 452 ms Wall time: 452 ms	0.236601 seconds (27.00 k allocations: 120.572 MiB, 6.44% compilation time)	0.473383 seconds (17.49 k allocations: 120.002 MiB, 45.32% gc time, 6.97% compilation time: 100% of which was recompilation) ?
			0.226617 seconds (3.08 k allocations: 118.958 MiB, 5.16% gc time)

PRINCETON UNIVERSITY

iris hep

AwkwardArray.jl Overhead

Compared with Using Typed Arrays

- Started with using AwkwardArray and compared it to a Julia native typed array: Vector

- **Takeaway**: no significant overhead seen after small changes to Julia main_looper code.

```
[12]: %%julia

      array = @time main_looper(events)

        0.459656 seconds (398.55 k allocations: 77.552 MiB, 2.98% gc time,
      75.63% compilation time)

[12]: 20525-element AwkwardArray.PrimitiveArray{Float64, Vector{Float64},
      :default}:
        125.12303161621094
        123.90653991699219
        124.15757751464844
        122.6549301147461
        125.26071166992188
        124.77593994140625
        124.20553588867188
        124.42249298095703
        110.03680419921875
        124.46846008300781
          ⋮
        127.15644836425781
         70.50875091552734
```

```
[30]: %%julia

      array = @time main_looper(events)

        0.225840 seconds (185.55 k allocations: 73.431 MiB, 2.74% gc time, 46.84% compilation time)

[30]: 20525-element Vector{Float64}:
        125.12303161621094
        123.90653991699219
        124.15757751464844
        122.6549301147461
        125.26071166992188
        124.77593994140625
        124.20553588867188
        124.42249298095703
        110.03680419921875
        124.46846008300781
          ⋮
```

ChatGPT

```
[38]: %%julia

      array = @time main_looper(events)

        0.257670 seconds (185.09 k allocations: 71.296 MiB, 1.61% gc time, 45.21% compilation time)

[38]: 20525-element AwkwardArray.PrimitiveArray{Float64, Vector{Float64}, :default}:
        125.12303161621094
        123.90653991699219
        124.15757751464844
        122.6549301147461
        125.26071166992188
        124.77593994140625
        124.20553588867188
        124.42249298095703
        110.03680419921875
        124.46846008300781
          ⋮
```

But can we do better?

PRINCETON UNIVERSITY

iris hep

Small Code Changes

in Destructure and Skip

```julia
function main_looper(events)
    array = AwkwardArray.PrimitiveArray{Float64}()
    for evt in events

        (; Muon_charge) = evt
        if length(Muon_charge) != 4
            continue
        end
        sum(Muon_charge) != 0 && continue # shortcut if-else

        (; Muon_pt, Muon_eta, Muon_phi, Muon_mass) = evt
        higgs_4vector = sum(LorentzVectorCyl.(Muon_pt, Muon_eta, Muon_phi, Muon_mass))
        higgs_mass = mass(higgs_4vector)

        push!(array, higgs_mass)
    end

    return array
end
```

```julia
function main_looper(events)
    # Create an empty AwkwardArray for storing the Higgs mass values
    array = AwkwardArray.PrimitiveArray{Float64}()

    # Loop over events and process only valid ones
    for evt in events
        # Destructure the necessary fields from the event
        (; Muon_charge, Muon_pt, Muon_eta, Muon_phi, Muon_mass) = evt

        # Skip event if it doesn't meet the required conditions
        if length(Muon_charge) != 4 || sum(Muon_charge) != 0
            continue
        end

        # Create Lorentz vectors for the muons and calculate the Higgs mass
        higgs_4vector = sum(LorentzVectorCyl.(Muon_pt, Muon_eta, Muon_phi, Muon_mass))
        higgs_mass = mass(higgs_4vector)

        # Add the result to the AwkwardArray
        push!(array, higgs_mass)
    end

    # Return the final AwkwardArray containing Higgs masses
    return array
end
```

ChatGPT

```
[52]: %%julia

      array = @time main_looper(events)

        0.145763 seconds (22.94 k allocations: 59.588 MiB, 2.19% gc time)

[52]: 20525-element AwkwardArray.PrimitiveArray{Float64, Vector{Float64}, :default}:
       125.12303161621094
       123.90653991699219
       124.15757751464844
       122.6549301147461
       125.26071166992188
```

- Time reduced from 0.459656 seconds to 0.145763 seconds

But can we do better?

Ensuring Type Stability

```julia
function main_looper(events::AwkwardArray.RecordArray)
    # Pre-allocate an AwkwardArray to store Higgs mass values
    array = AwkwardArray.PrimitiveArray{Float64}(undef, length(events))
    count = 0  # To track valid entries

    for evt in events
        # Destructure the necessary fields from the event with concrete types
        (; Muon_charge::Vector{Float64}, Muon_pt::Vector{Float64}, Muon_eta::Vector{Float64},
           Muon_phi::Vector{Float64}, Muon_mass::Vector{Float64}) = evt

        # Check conditions with inlined logic
        if length(Muon_charge) != 4 || sum(Muon_charge) != 0
            continue
        end

        # Compute the Lorentz vector sum and Higgs mass
        higgs_4vector = sum(LorentzVectorCyl.(Muon_pt, Muon_eta, Muon_phi, Muon_mass))
        higgs_mass = mass(higgs_4vector)

        # Store the result in the pre-allocated array
        count += 1
        array[count] = higgs_mass
    end

    # Resize the array to only include valid entries
    return AwkwardArray.subarray(array, 1:count)
end
```

ChatGPT

```
[56]: %%julia

      array = @time main_looper(events)

        0.119881 seconds (22.94 k allocations: 59.588 MiB, 3.07% gc time)
[56]: 20525-element AwkwardArray.PrimitiveArray{Float64, Vector{Float64}, :default}:
       125.12303161621094
       123.90653991699219
       124.15757751464844
       122.6549301147461
       125.26071166992188
       124.77593994140625
       124.20553588867188
       124.42249298095703
       110.03680419921875
       124.46846008300781
```

- Avoid unnecessary recompilation!

But can we do better?

Multithreading Support

Experimental in JuliaCall

```
[89]: %%julia

using AwkwardArray
using Base.Threads

function main_looper_awkward(events)
    array = AwkwardArray.PrimitiveArray{Float64}()
    lock_obj = ReentrantLock()  # Create a lock object to control access to shared array

    @threads for i in 1:length(events)
        evt = events[i]

        # Destructure the necessary fields from the event
        (; Muon_charge, Muon_pt, Muon_eta, Muon_phi, Muon_mass) = evt

        # Skip event if it doesn't meet the required conditions
        if length(Muon_charge) != 4 || sum(Muon_charge) != 0
            continue
        end

        # Create Lorentz vectors for the muons and calculate the Higgs mass
        higgs_4vector = sum(LorentzVectorCyl.(Muon_pt, Muon_eta, Muon_phi, Muon_mass))
        higgs_mass = mass(higgs_4vector)

        # Use lock to safely push! into the shared array
        lock(lock_obj)  # Explicitly lock before modifying shared data
        try
            push!(array, higgs_mass)
        finally
            unlock(lock_obj)  # Ensure the lock is always released
        end
    end

    return array
end
```

ChatGPT

```
% export JULIA_NUM_THREADS=4
```

```
% export PYTHON_JULIACALL_HANDLE_SIGNALS=yes
```

- **Execution time reduced by 88%**, speeding up from 0.5 seconds to 0.06 seconds—a **8.33x performance improvement**.
- **Memory usage optimized**, cutting allocations from 398k to 24k, making the process much more efficient.
- Overall, the code is **much faster and leaner**, showing significant gains in both speed and memory management.

```
[91]: %%julia

array = @time main_looper_awkward(events)

  0.063811 seconds (24.00 k allocations: 87.739 MiB)
[91]: 20525-element AwkwardArray.PrimitiveArray{Float64, Vector{Float64}, :default}:
 125.12303161621094
 123.90653991699219
 124.15757751464844
```

Performance Scaling

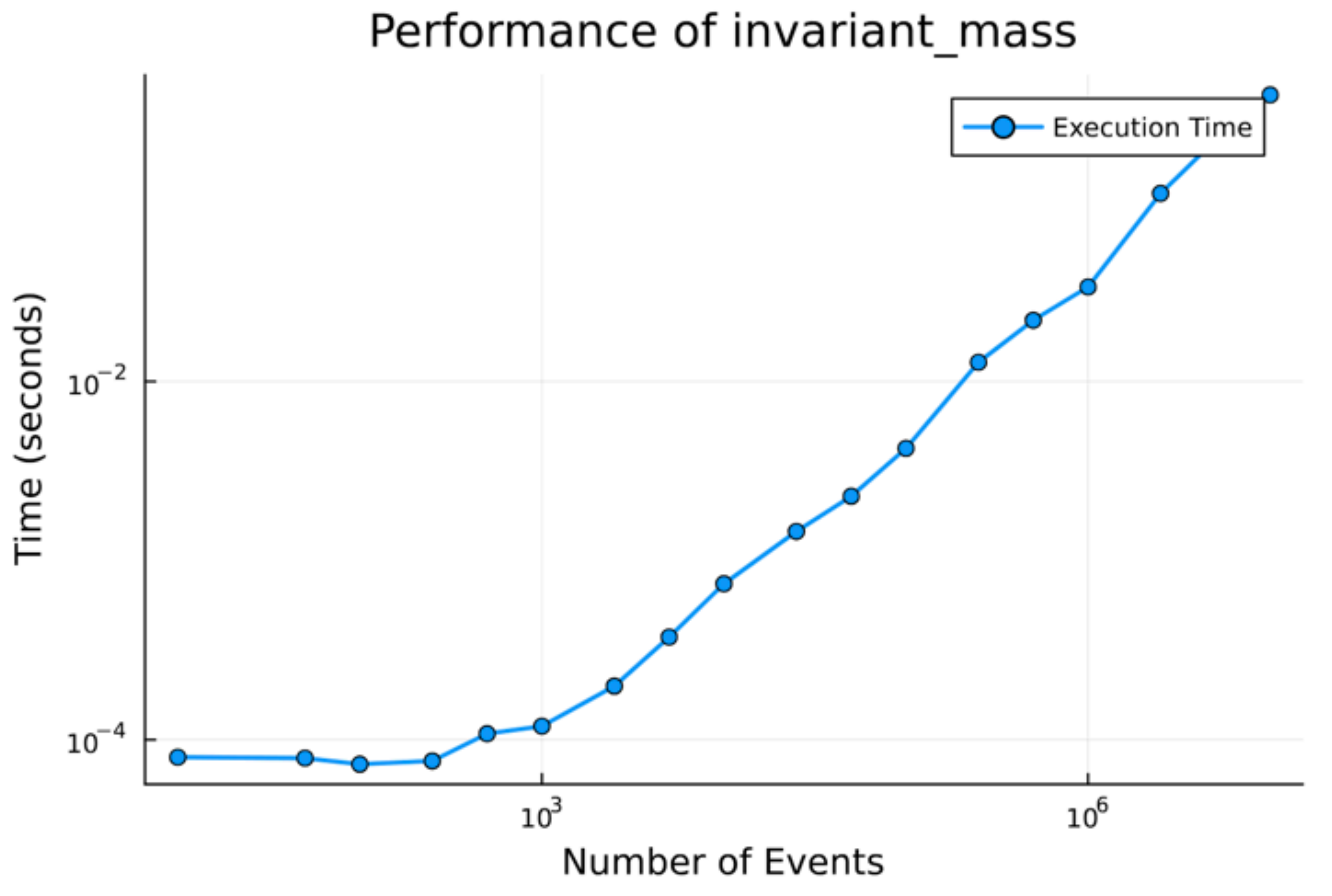
Increasing # Events

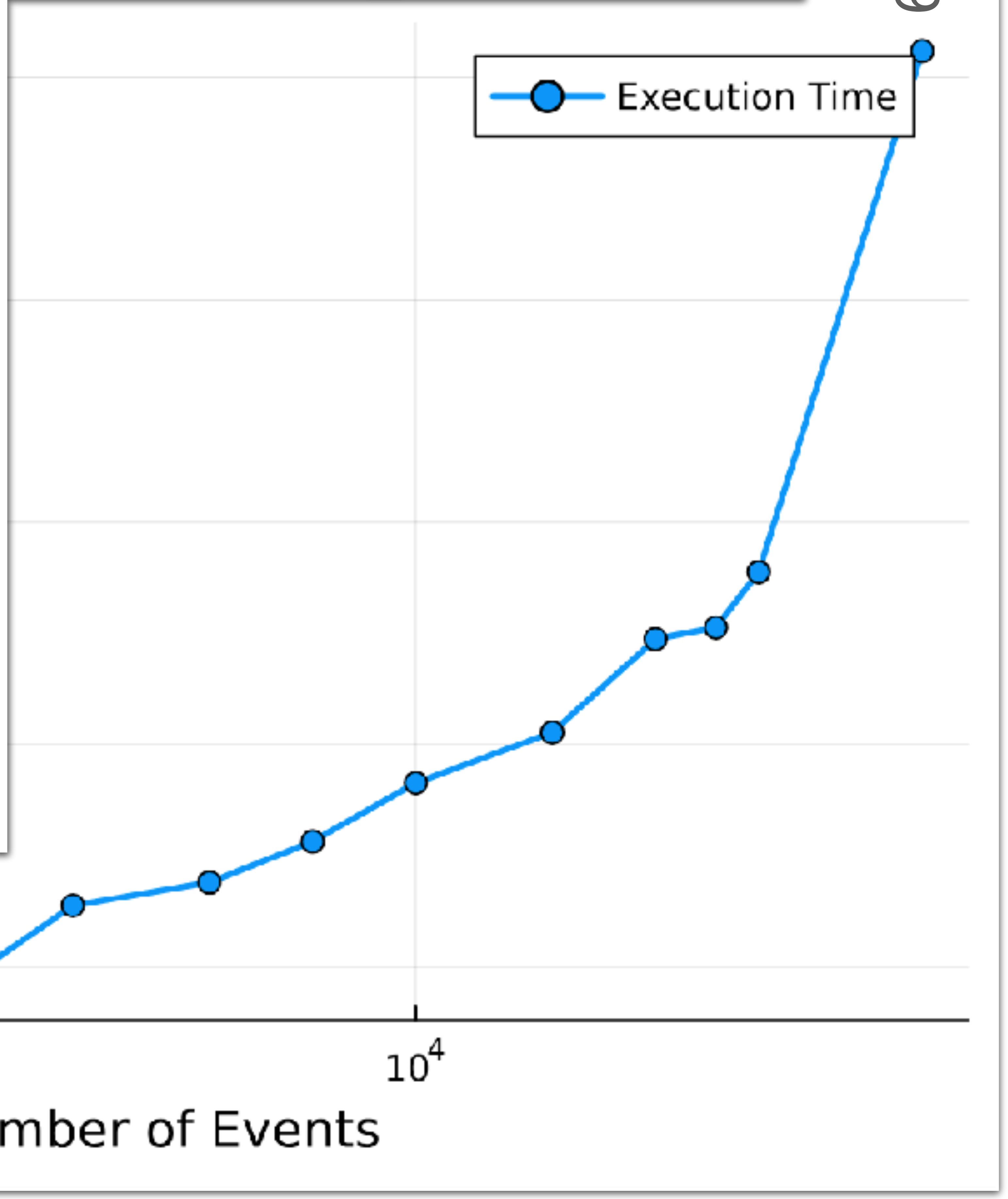
```
[111]: %%julia

       @time main_looper(events[1:10000])

         0.054722 seconds (213.33 k allocations: 65.210 MiB)

[111]: 696-element AwkwardArray.PrimitiveArray{Float64, Vector
        125.12303161621094
        123.90653991699219
        124.15757751464844
        125.54163360595703
        123.40087890625
        100.83844757080078
        102.18019104003906
        124.19779968261719
        124.63880920410156
        124.16439819335938
          ⋮
```

```
[113]: %%julia

       @time main_looper(events[1:100000])

         0.216400 seconds (2.11 M allocations: 305.127 MiB)

[113]: 6785-element AwkwardArray.PrimitiveArray{Float64, Vector
        126.94151306152344
        126.7888412475586
        125.12303161621094
        125.17405700683594
        121.99496459960938
        123.90653991699219
        126.61412811279297
        124.15757751464844
        105.9798583984375
        125.23162078857422
          ⋮
```

```
@time main_looper(events)
  9.897901 seconds (2.09 M allocations: 15.559 GiB, 16.83% gc time)
1915776-element AwkwardArray.PrimitiveArray{Float64, Vector{Float64}, :default}:
```

61,540,413

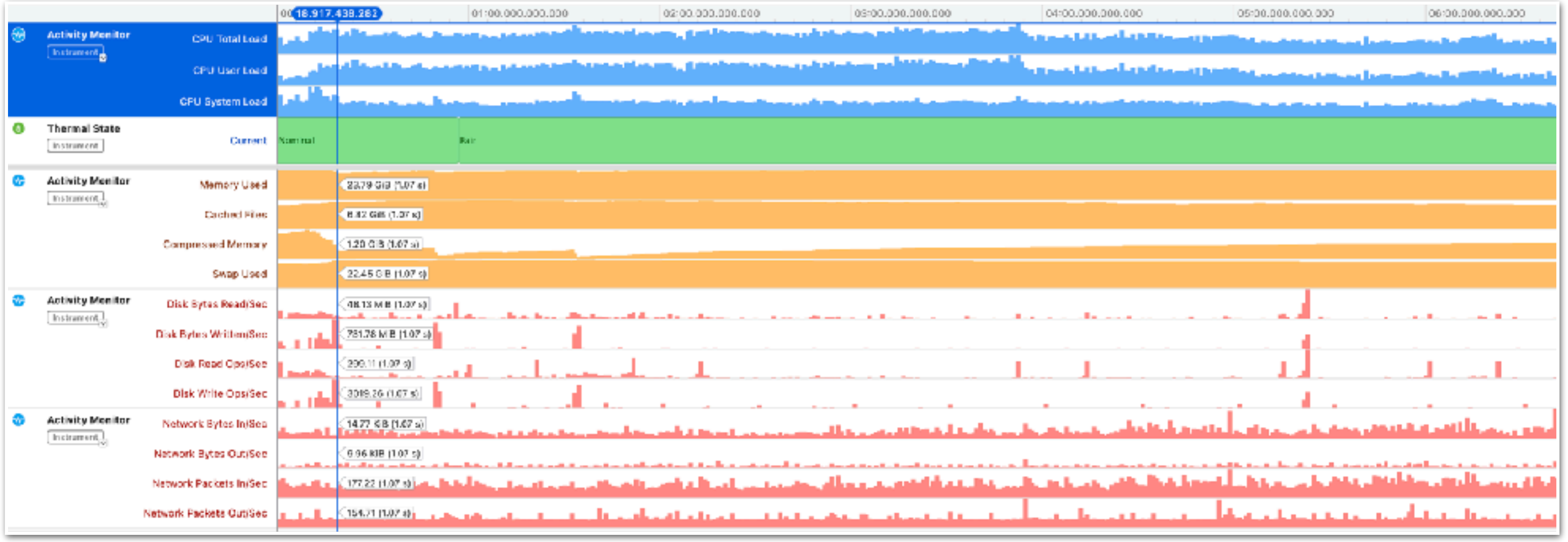


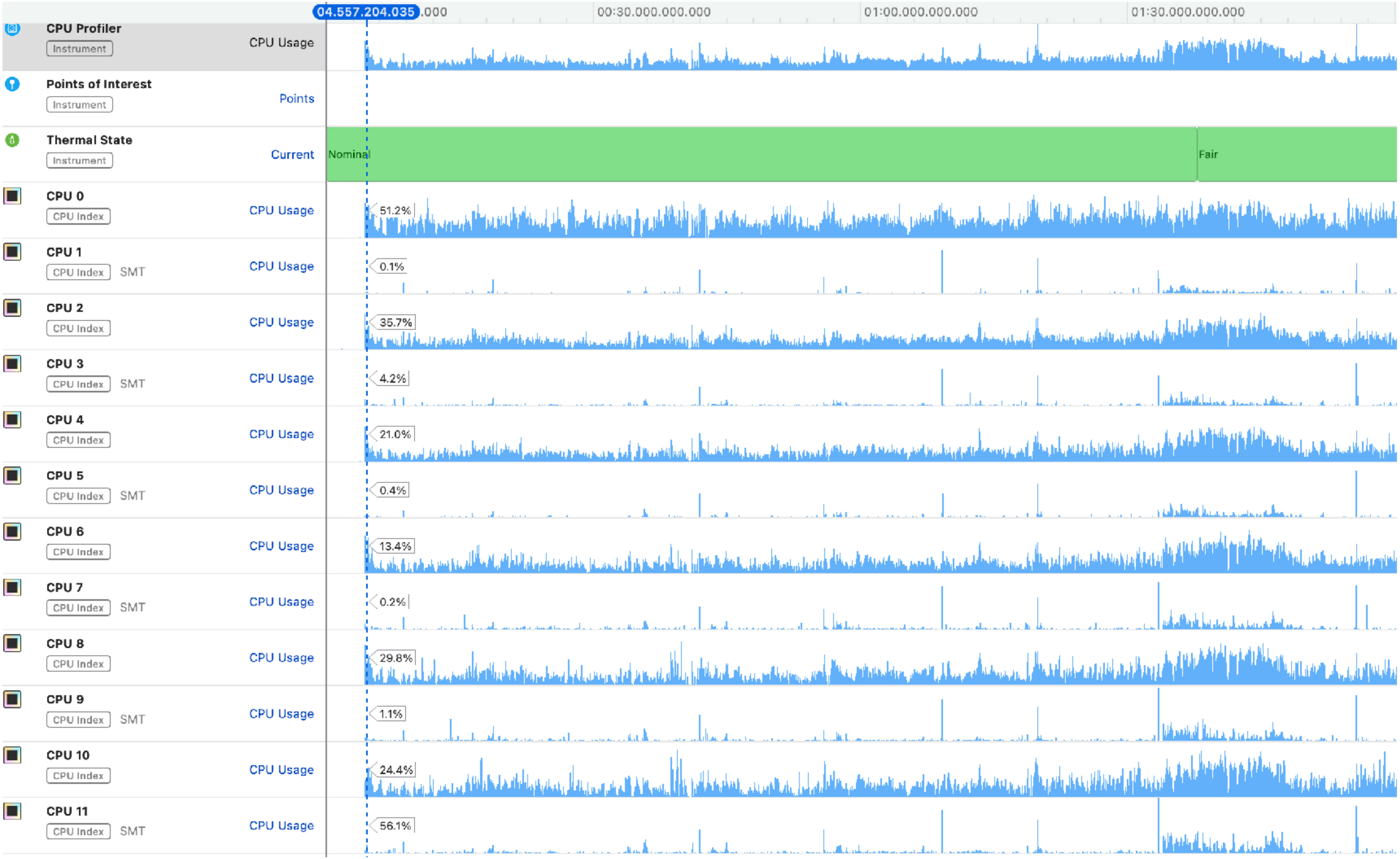


Instruments

Activity Monitor

- macOS Big Sur version 11.6
- Processor 2.6 GHz 6-Core Intel Core i7
- Memory 32 GB 2667 MHz DDR4




04.557.204.035
.000
00:30.000.000.000
01:00.000.000.000
01:30.000.000.000
CPU Profiler
Instrument
CPU Usage
Points of Interest
Instrument
Points
Thermal State
Instrument
Current
Nominal
Fair
CPU 0
CPU Index
CPU Usage
51.2%
CPU 1
CPU Index
SMT
CPU Usage
0.1%
CPU 2
CPU Index
CPU Usage
35.7%
CPU 3
CPU Index
SMT
CPU Usage
4.2%
CPU 4
CPU Index
CPU Usage
21.0%
CPU 5
CPU Index
SMT
CPU Usage
0.4%
CPU 6
CPU Index
CPU Usage
13.4%
CPU 7
CPU Index
SMT
CPU Usage
0.2%
CPU 8
CPU Index
CPU Usage
29.8%
CPU 9
CPU Index
SMT
CPU Usage
1.1%
CPU 10
CPU Index
CPU Usage
24.4%
CPU 11
CPU Index
SMT
CPU Usage
56.1%

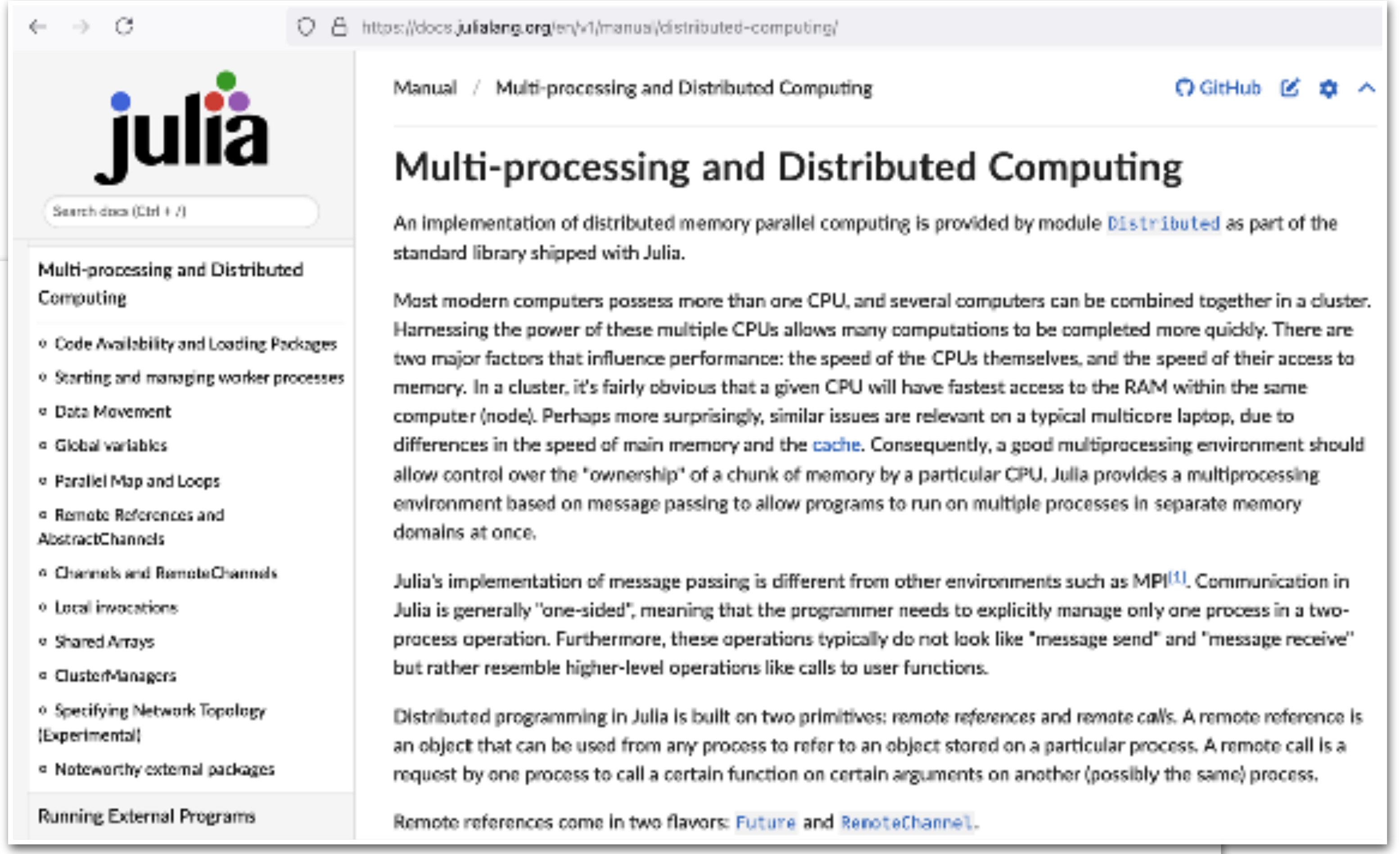
PRINCETON UNIVERSITY

But can we do better?

Multi-processing and Distributed Computing

```
[7]: %%julia

     @everywhere include("DummyAwkwardModule.jl")

[8]: %%julia

     using .DummyAwkwardModule

[9]: %%julia

     MuMu = @spawnat :any invariant_mass(events);

[11]: %%julia

     @time fetch(MuMu)

       0.000005 seconds

[11]: 5638149-element AwkwardArray.PrimitiveArray{Float64, Vector{Float64}, :default
       113.64686584472656
        88.29710388183594
        88.33483123779297
        91.27149963378906
        93.55725860595703
        90.91211700439453
        89.15238952636719
        82.29732513427734
        94.57678985595703
        89.23975372314453
         ⋮
```

```julia
module DummyAwkwardModule

export invariant_mass

using LorentzVectorHEP, AwkwardArray
using UnROOT

using Base.Threads

function invariant_mass(cms_events)

    array = AwkwardArray.PrimitiveArray{Float64}()
    lock_obj = ReentrantLock()  # Create a lock object to control access to shared array

    @threads for i in 1:length(cms_events)
        evt = cms_events[i]

        # Destructure the necessary fields from the event
        (; Muon_charge, Muon_pt, Muon_eta, Muon_phi, Muon_mass, nMuon) = evt

        # Skip event if it doesn't meet the required conditions
        if nMuon != 2 || Muon_charge[1] == Muon_charge[2]
            continue
        end

        # Calculate invariant mass using LorentzVectorHEP for clarity and accuracy
        muon1 = LorentzVectorCyl(Muon_pt[1], Muon_eta[1], Muon_phi[1], Muon_mass[1])
        muon2 = LorentzVectorCyl(Muon_pt[2], Muon_eta[2], Muon_phi[2], Muon_mass[2])
        result = mass(muon1 + muon2)

        # Only add masses greater than 70 GeV
        if result > 70
            # Use lock to safely push! into the shared array
            lock(lock_obj)  # Explicitly lock before modifying shared data
            try
```



https://docs.julialang.org/en/v1/manual/distributed-computing/

julia

Search docs (Ctrl + /)

Multi-processing and Distributed Computing
- Code Availability and Loading Packages
- Starting and managing worker processes
- Data Movement
- Global variables
- Parallel Map and Loops
- Remote References and AbstractChannels
- Channels and RemoteChannels
- Local invocations
- Shared Arrays
- ClusterManagers
- Specifying Network Topology (Experimental)
- Noteworthy external packages

Running External Programs

Manual / Multi-processing and Distributed Computing GitHub

Multi-processing and Distributed Computing

An implementation of distributed memory parallel computing is provided by module `Distributed` as part of the standard library shipped with Julia.

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly. There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node). Perhaps more surprisingly, similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache. Consequently, a good multiprocessing environment should allow control over the "ownership" of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

Julia's implementation of message passing is different from other environments such as MPI[1]. Communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one process in a two-process operation. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions.

Distributed programming in Julia is built on two primitives: *remote references* and *remote calls*. A remote reference is an object that can be used from any process to refer to an object stored on a particular process. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

Remote references come in two flavors: `Future` and `RemoteChannel`.

Summary

Optimizing Performance with Julia

- While we may not see a significant speedup from replacing NumPy, Awkward, or Numba with Julia in vectorized operations, identifying tasks that don't fit well with these libraries can unlock Julia's true potential.
- Developing custom kernels for specific problems may lead to innovative solutions, even if it’s not immediately obvious.
- Despite challenges in multilingual runtime environments and experimental thread support, the ongoing evolution of Julia offers exciting opportunities for performance enhancement.