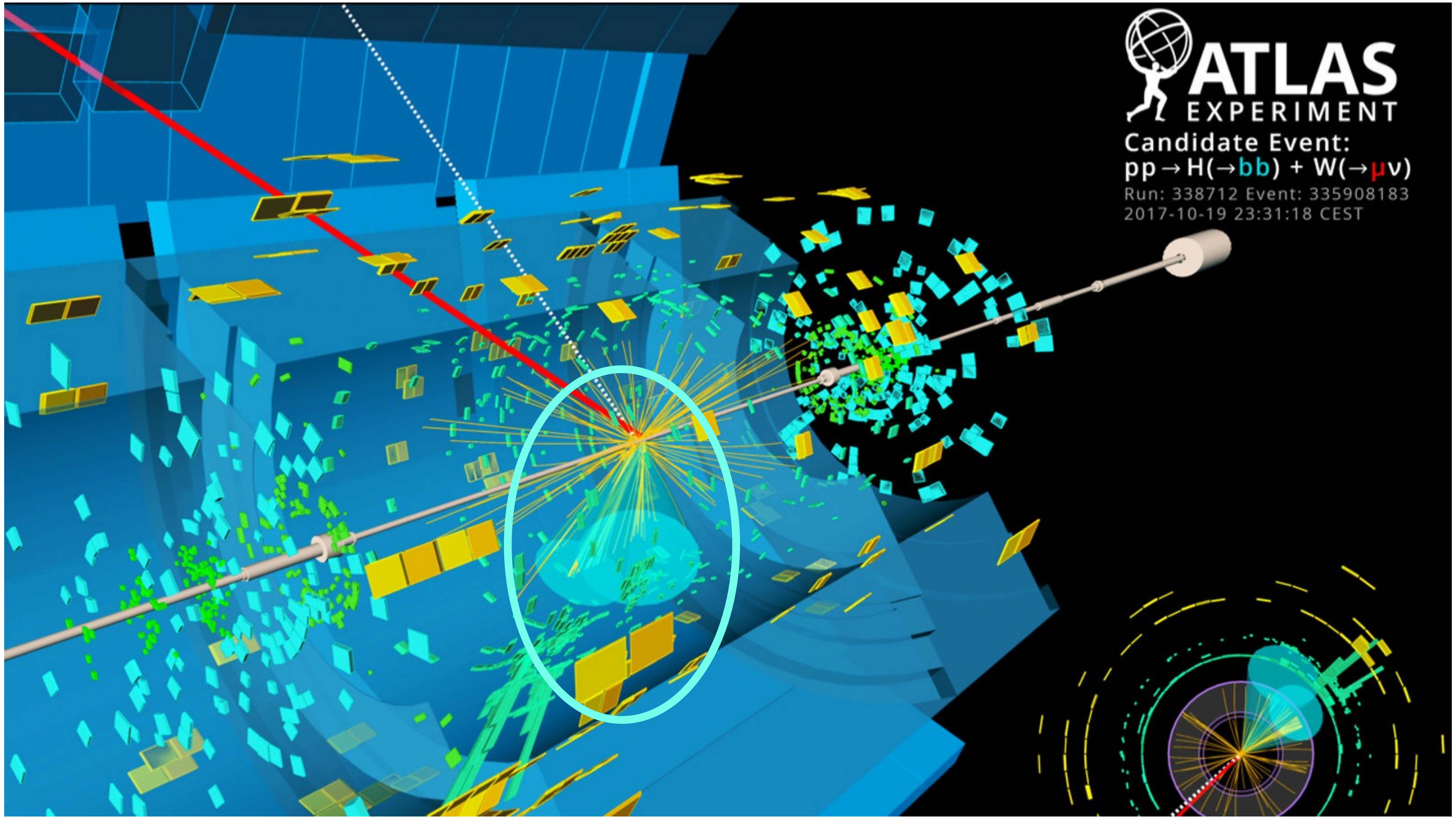
Fast Jet Finding in Julia

CHEP 2024

Graeme Stewart, Philippe Gras, Atell Krasnopolski, Sanmay Ganguly, Sattwamo Ghosh

EP-SFT

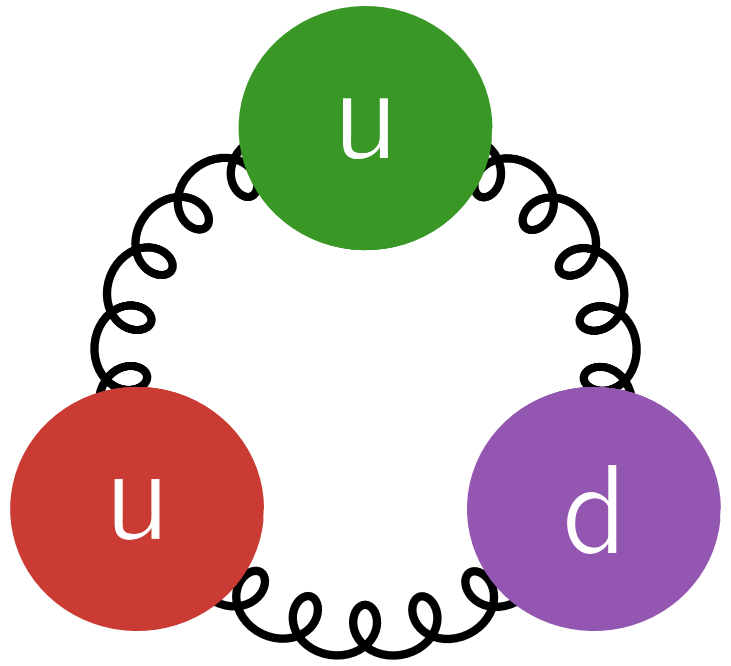
Jets at the LHC



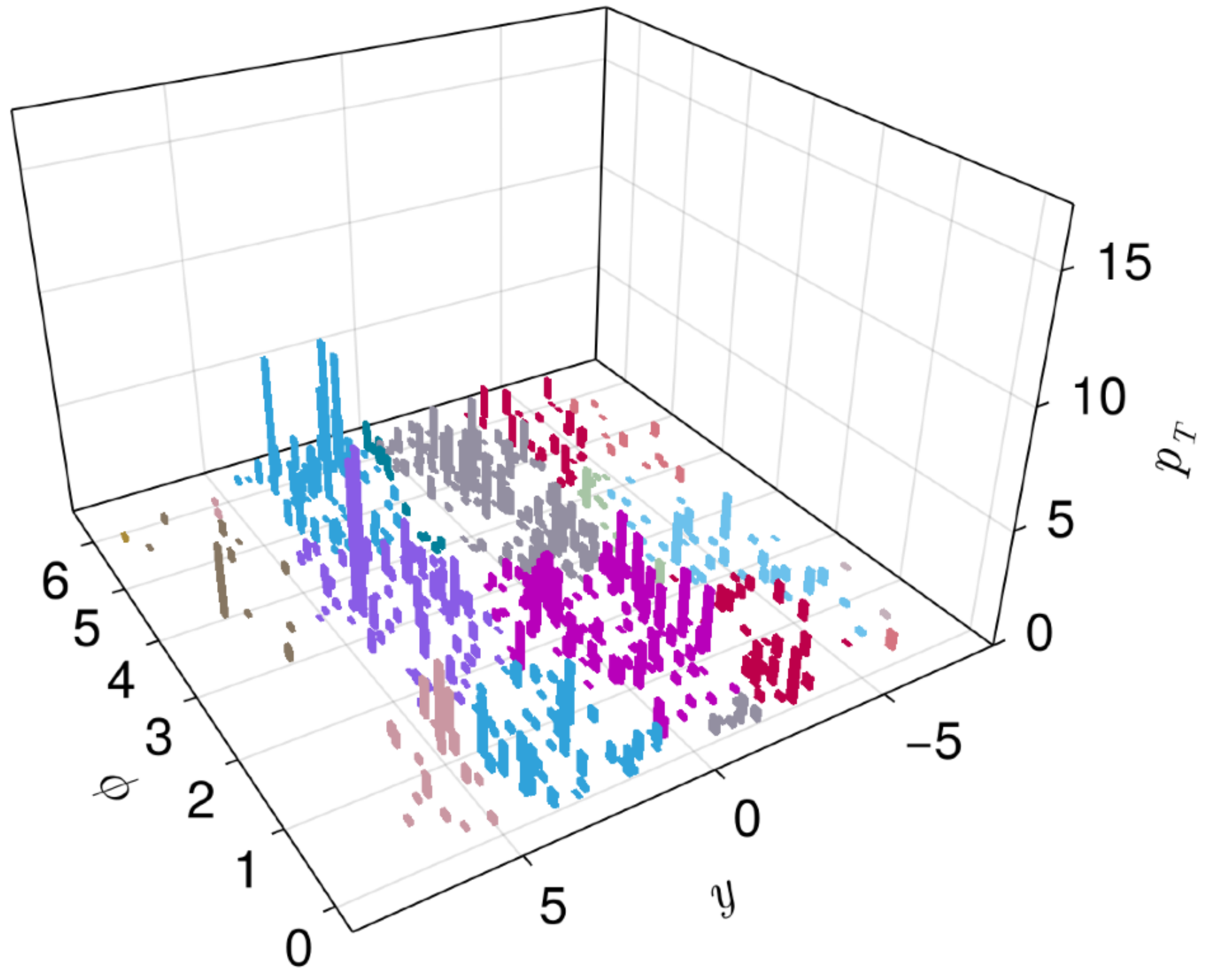
- Jets are a critical part of LHC physics
- Decays captured in the calorimeter
- Early stage of reconstruction to aggregate calo hits to jets

Jets in Julia…?

- There is a ubiquitously used jet finding package in C++, FastJet
- The initial motivation for trying to implement this in Julia was to investigate both performance and ergonomics
  - Presented at CHEP 2023 (comparing Julia, Python, accelerated Python and Fast jet)
  - Initial Julia results were very encouraging [arXiv:2309.17309]
    - Excellent runtime performance, easy to work with the code
- Decided to go ahead and make this a production Julia package
  - Meshes very well with other developments in the JuliaHEP universe



Sequential Jet Algorithms in Brief (pp flavour)



1. Define a distance parameter R (we use 0.4, which at LHC is typical)
    1. This is a "cone size"
2. For each active pseudo-jet i (=particle, cluster)
    1. Measure the geometric distance, d, to the nearest active pseudo-jet j, (if $d < R$ else $d = R$)
    2. Define the metric distance, d_{ij}, as

$$d_{ij} = d \times \min(p_{Ti}^{2p}, p_{Tj}^{2p})$$

3. Choose the jet with the lowest d_{ij}
    1. If this jet has an active partner j, merge these jets
    2. If not, this is a final jet
4. Repeat steps 2-3 until no jets remain active

There is a parallelisation opportunity here

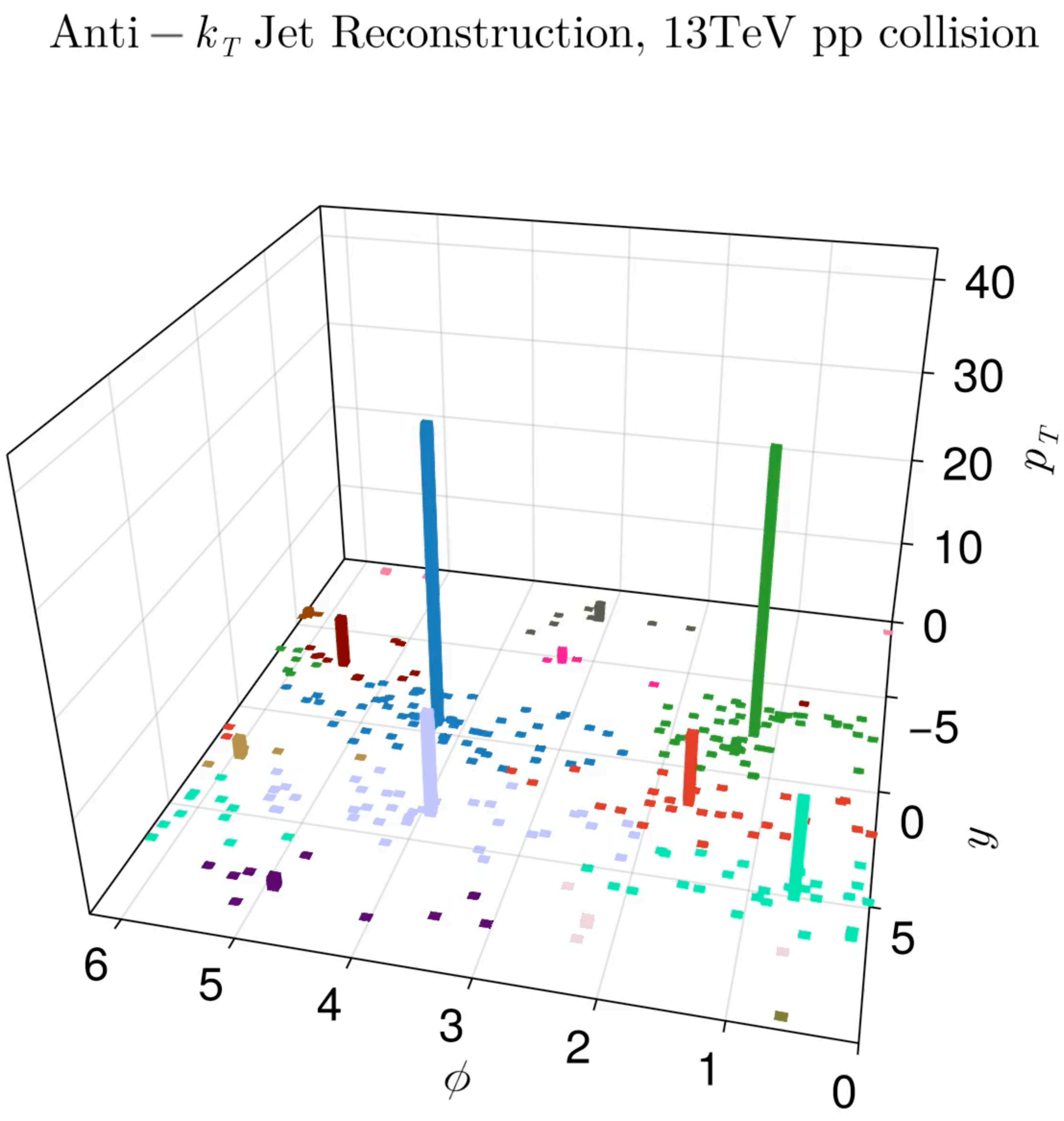
Algorithm:
p=-1 AntiKt
p=0 Cambridge/Achen
p=1 Inclusive Kt

This piece is serial(ish)

This algorithm from FastJet [arXiv:1111.6097]

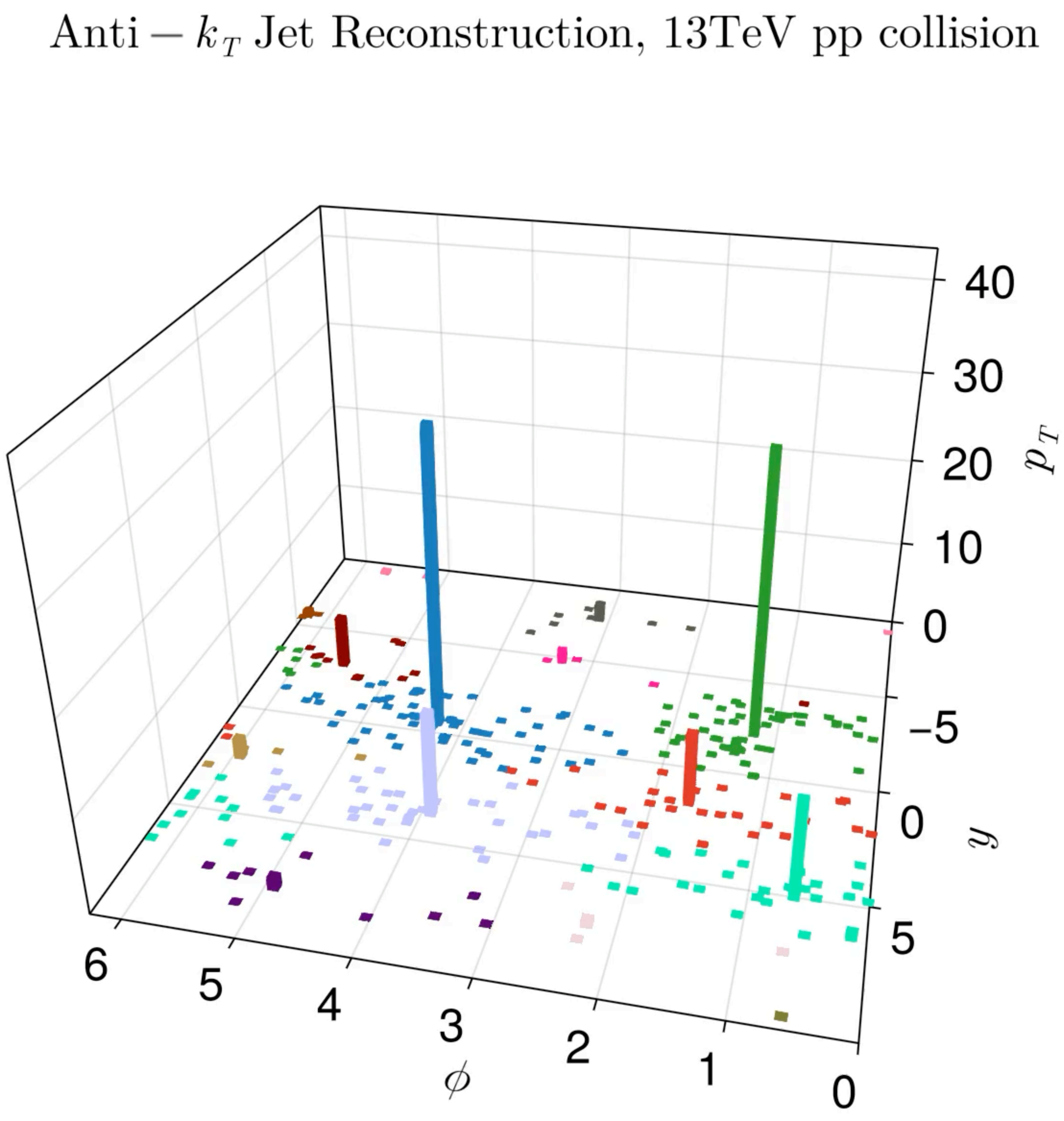
Anti-kT Example

- When $p = -1$ the algorithm favours merging of high p_T pseudo jets
- This provides infrared stability and co-linear safety
- Thus extremely popular algorithm at LHC



Anti-kT Example

- When $p = -1$ the algorithm favours merging of high p_T pseudo jets
- This provides infrared stability and co-linear safety
- Thus extremely popular algorithm at LHC

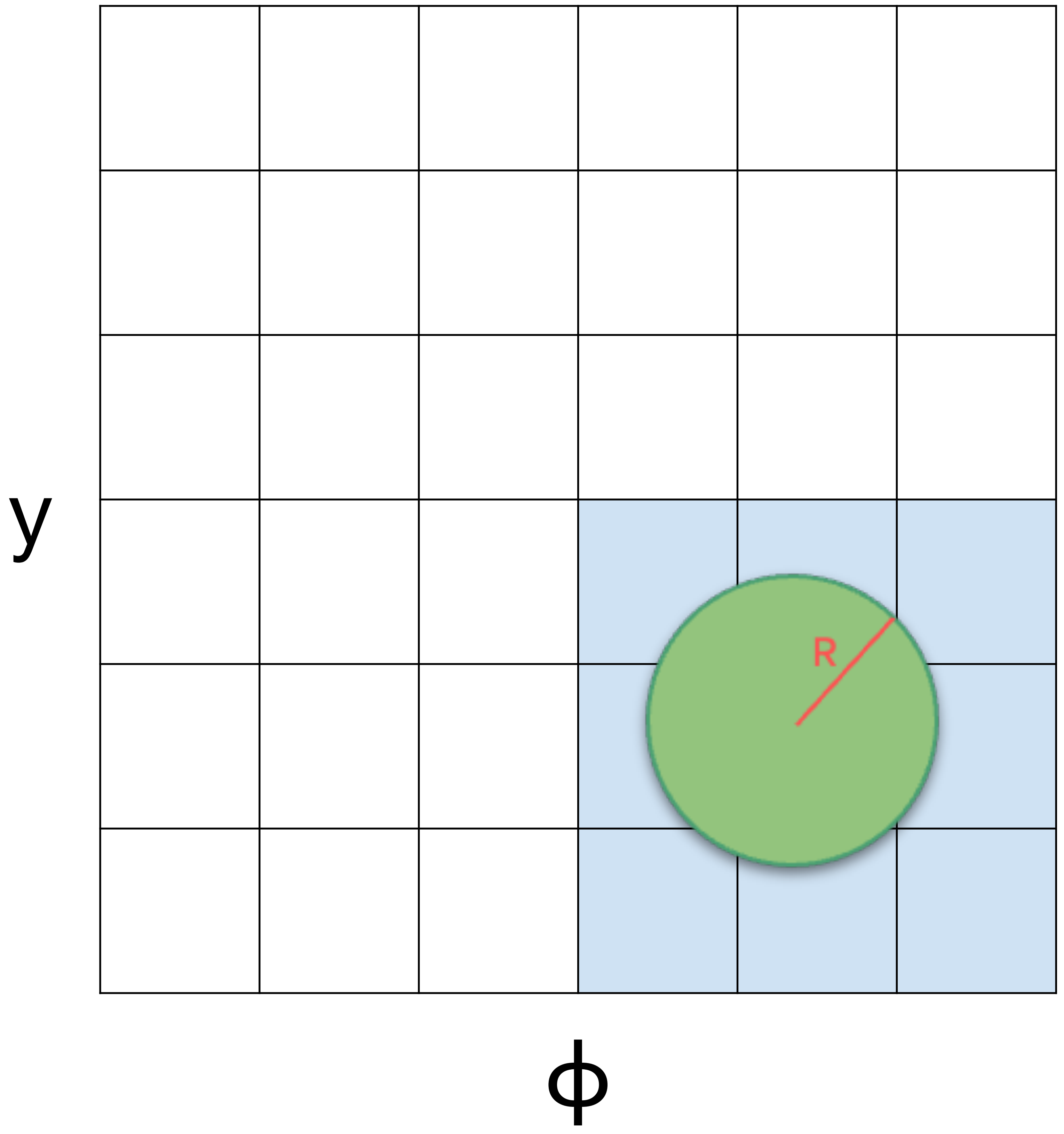


Different Strategies

- There are two implementation strategies
  - **N2Plain**: A basic implementation of the algorithm, essentially just implementing the flow on the previous slide, all jets considered in a global pool
  - **N2Tiled**: A tiled implementation of the algorithm, where the (y, ϕ) plane is split into tiles of size R
    - So that only neighbouring tiles need to be considered when calculating distances
- The tiled algorithm involves more bookkeeping, but reduces the work needing done
- The basic algorithm does more calculations, but these are more amenable to parallelisation

Tiled Implementation

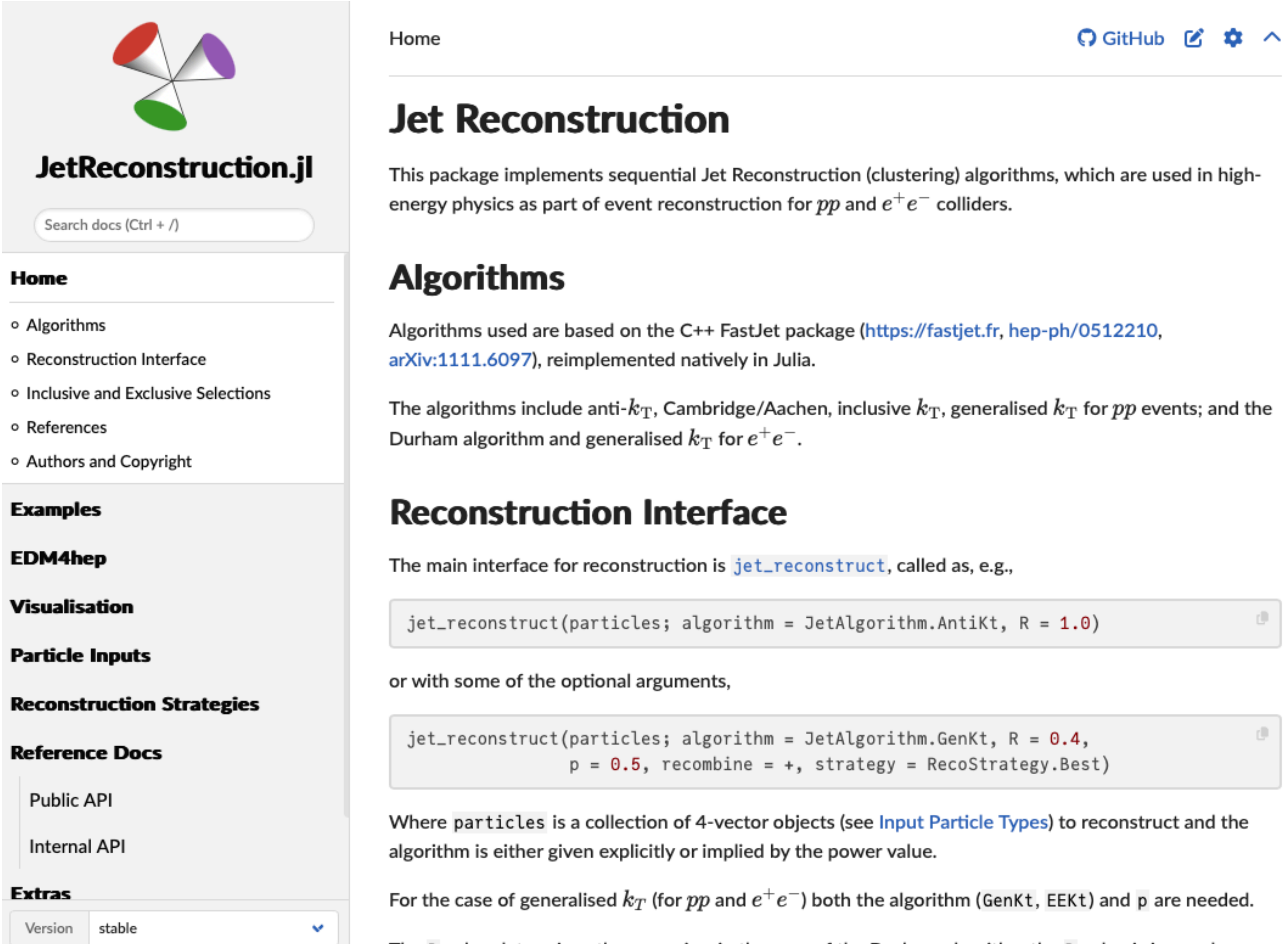
For a jet centred in the circle, only blue tile neighbours need to be considered



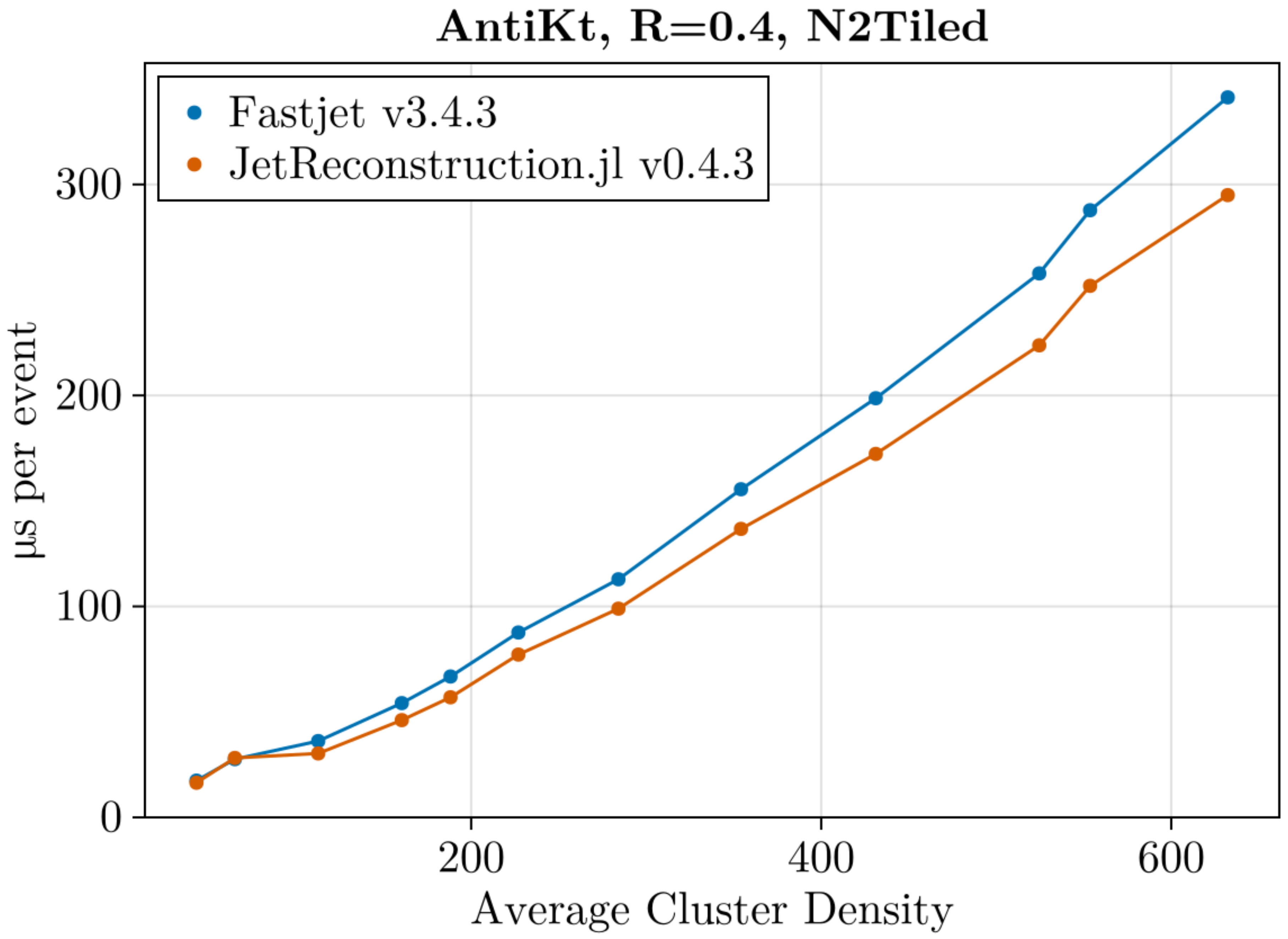
y = Rapidity

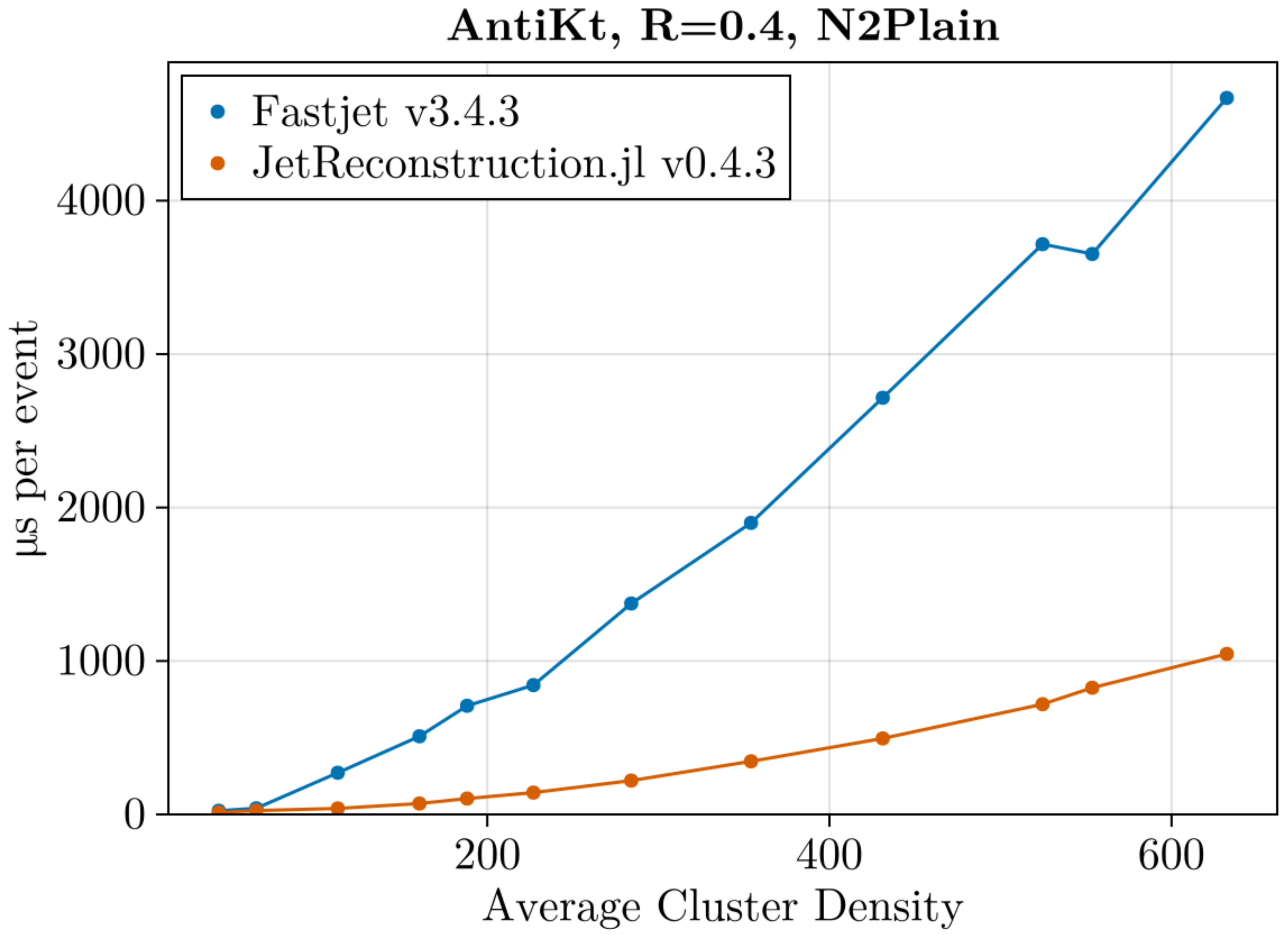
A Real Release!

- JetReconstruction.jl was release on June 17 this year (v0.3.0)
- A fair amount of refactoring was required to ensure that the two pp strategies (N2Plain and N2Tiled) behaved in the same way
  - Internal restructuring to uniformly use PseudoJets and return ClusterSequence objects
- Implemented *exclusive jet selections* (`n_jet` or `dij_max` cut)
- Implemented generalised kT algorithm (i.e. p_T^{2p} for arbitrary p)
- Choice of strategies: N2Plain, N2Tiled and **Best**
- Fixes to visualisation and improved examples
- Significant improvements to documentation
  - Documenter.jl setup
  - Published at https://juliahep.github.io/JetReconstruction.jl/stable/



pp Algorithm Performance*

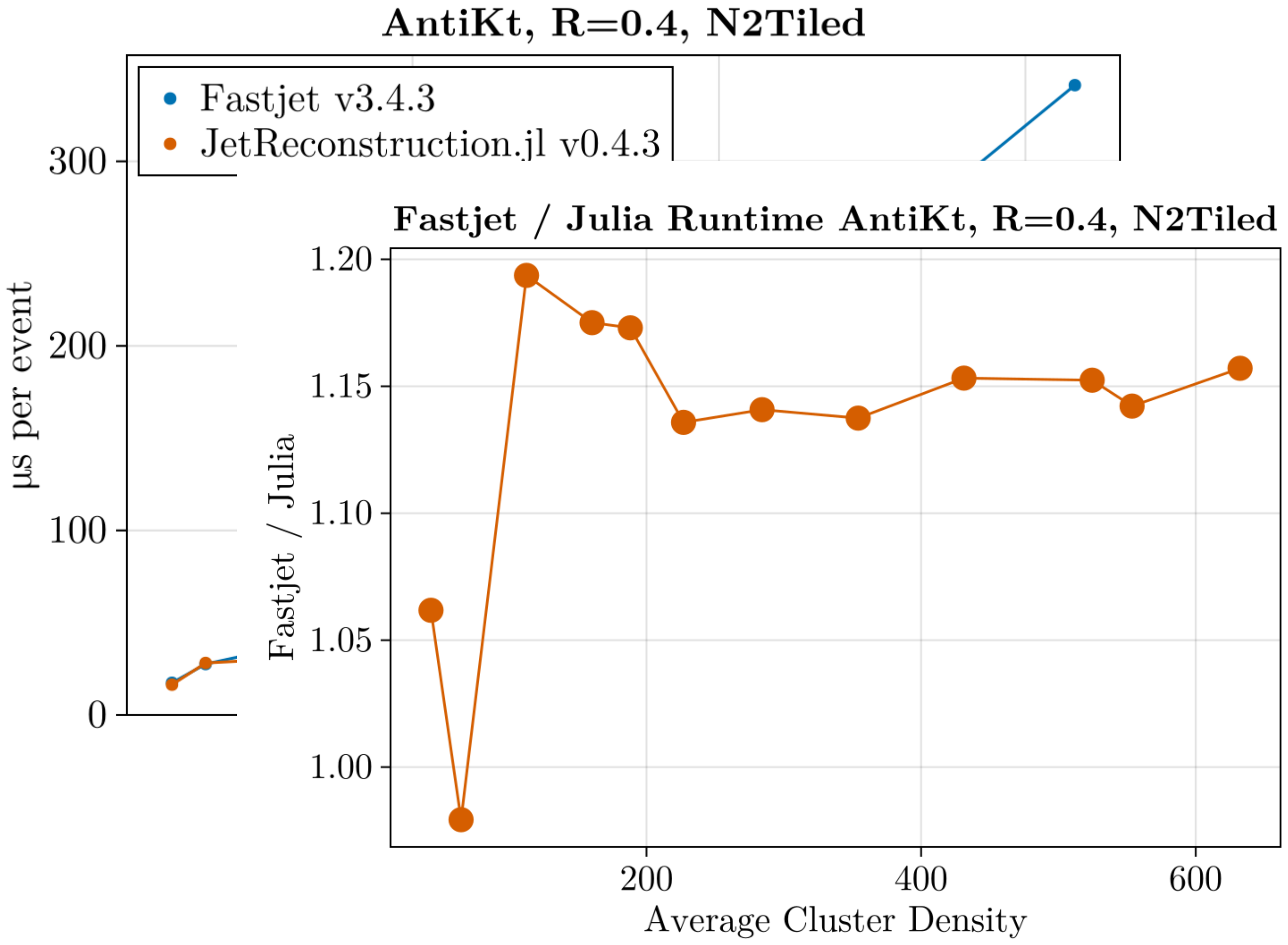


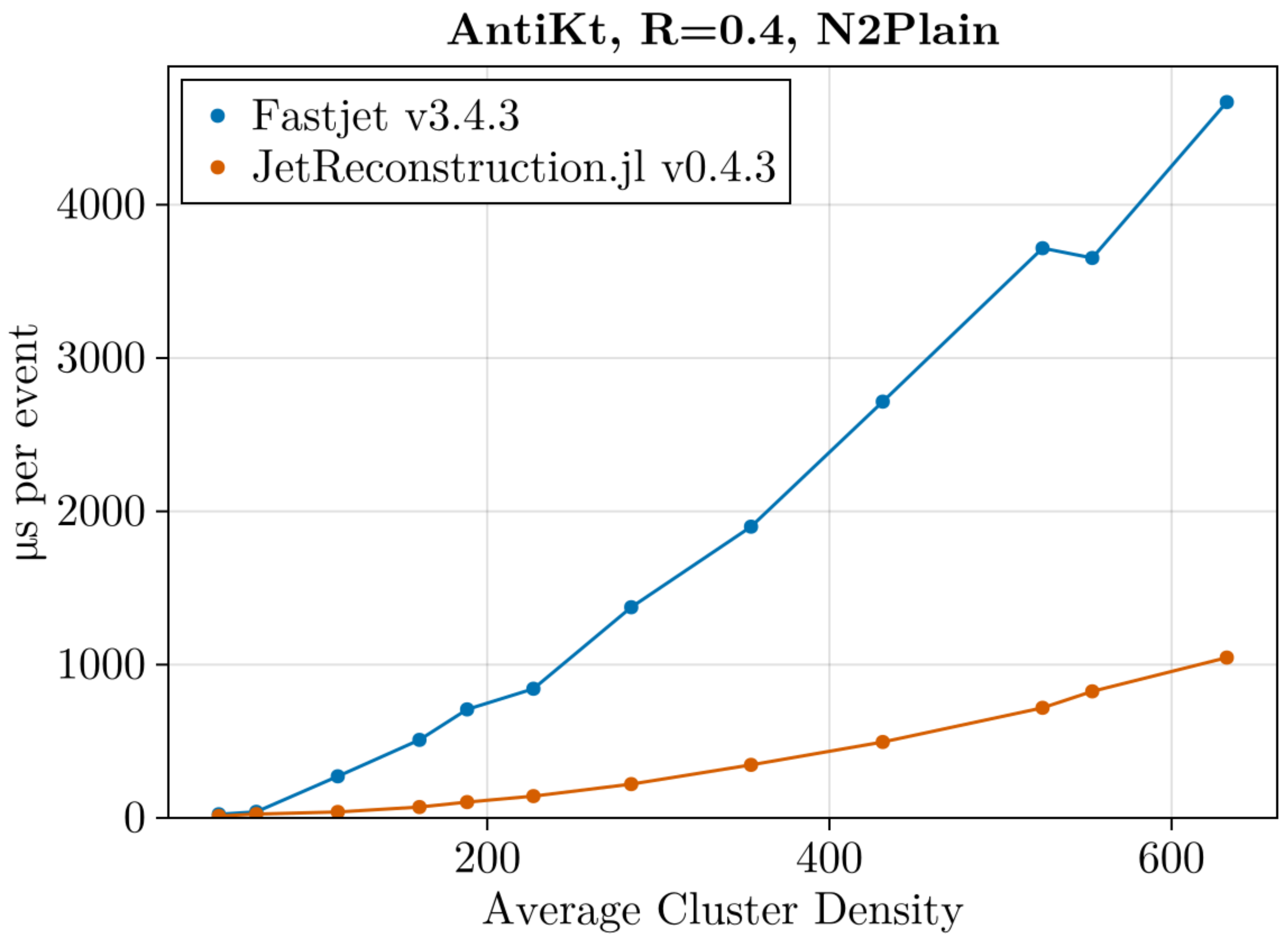


- Julia is consistently faster than Fastjet
  - N2Tiled: Gains are roughly 15% for the tiled algorithm
  - N2Plain: Fastjet for the plain algorithm is a bit pathological at low R - it's better behaved for $R \geq 1.0$
- Julia wins by taking more advantage of SIMD and loop vectorisation

* See backup for benchmark parameters

pp Algorithm Performance*

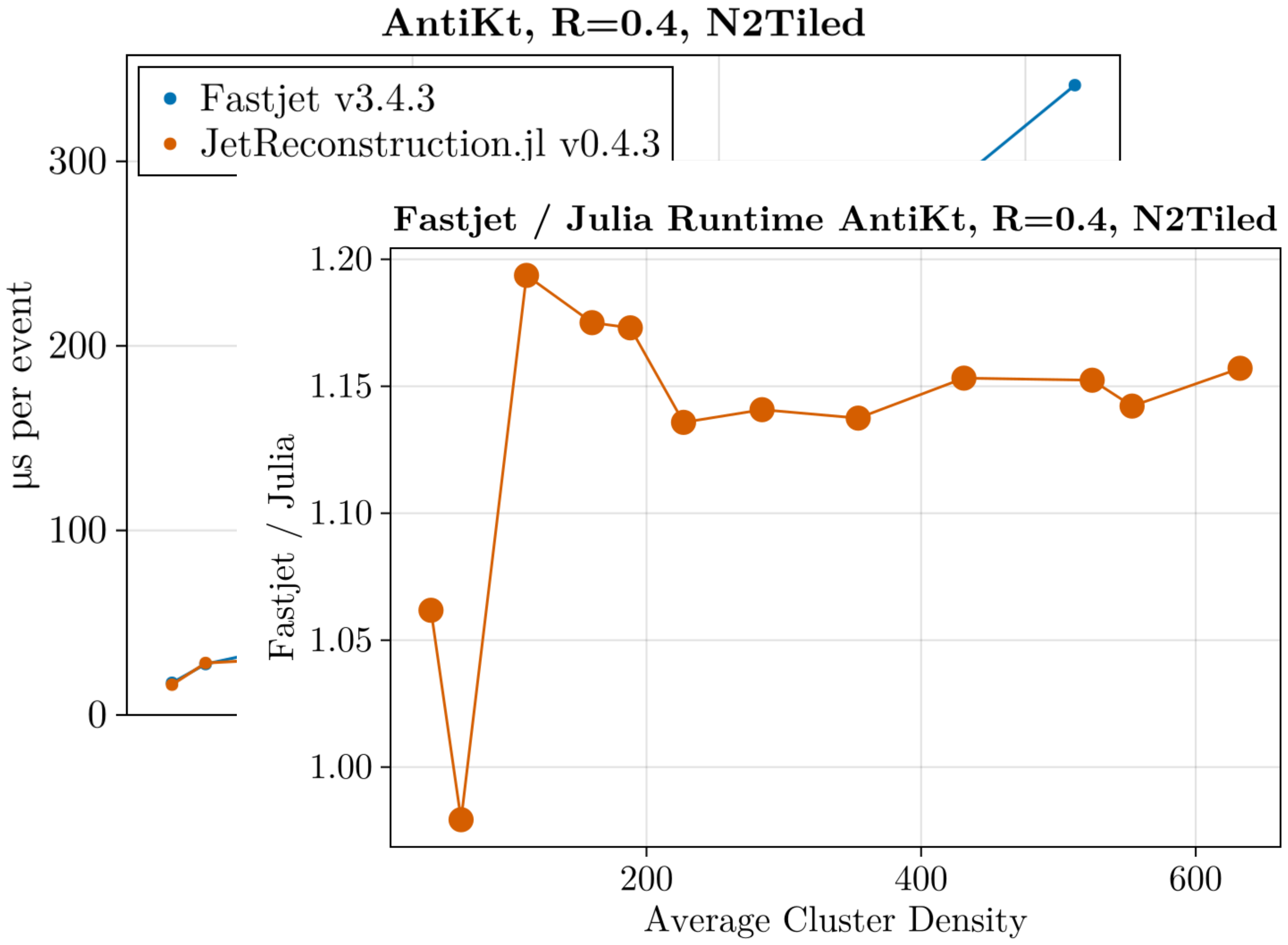


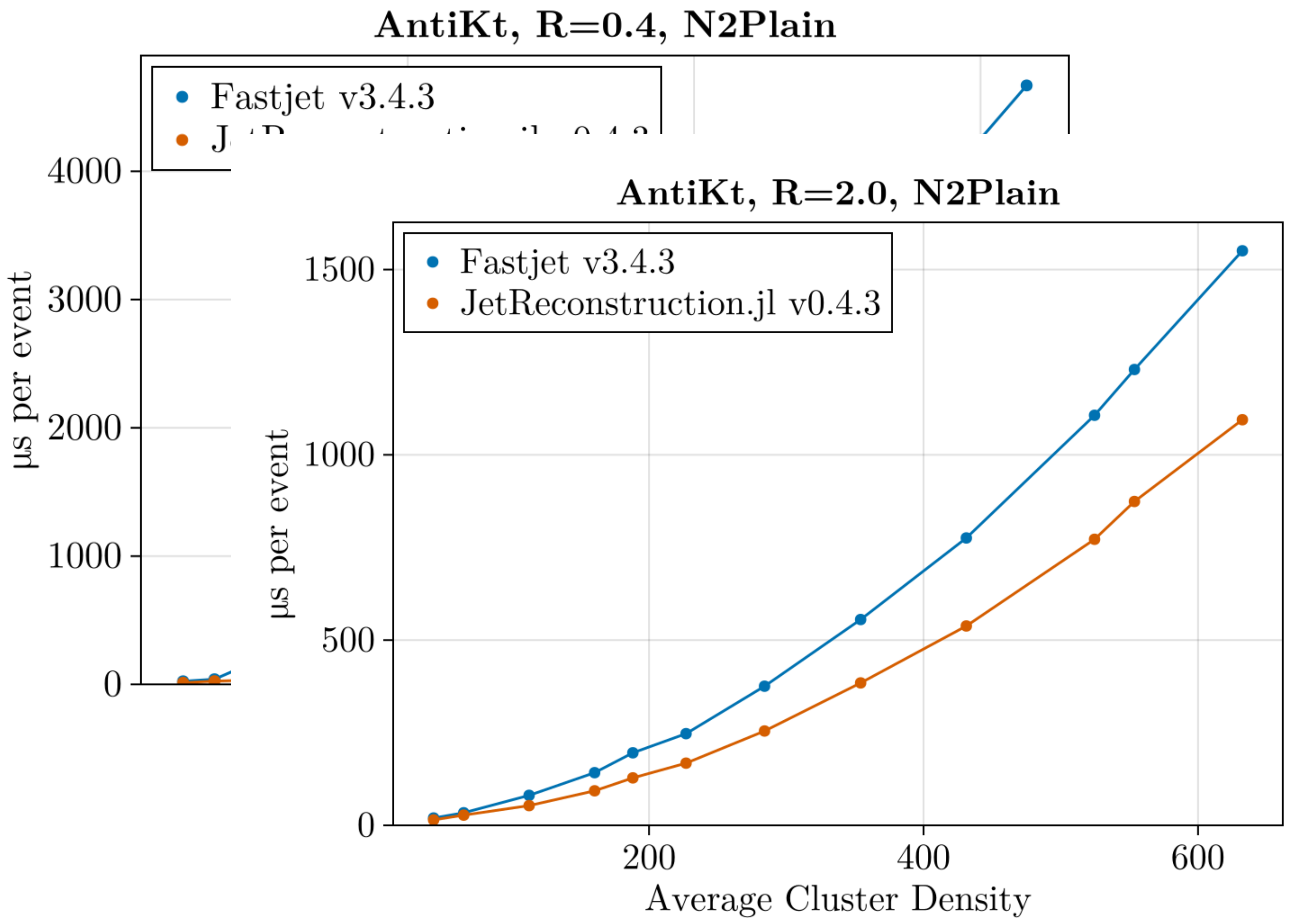


- Julia is consistently faster than Fastjet
  - N2Tiled: Gains are roughly 15% for the tiled algorithm
  - N2Plain: Fastjet for the plain algorithm is a bit pathological at low R - it's better behaved for $R \geq 1.0$
- Julia wins by taking more advantage of SIMD and loop vectorisation

* See backup for benchmark parameters

pp Algorithm Performance*

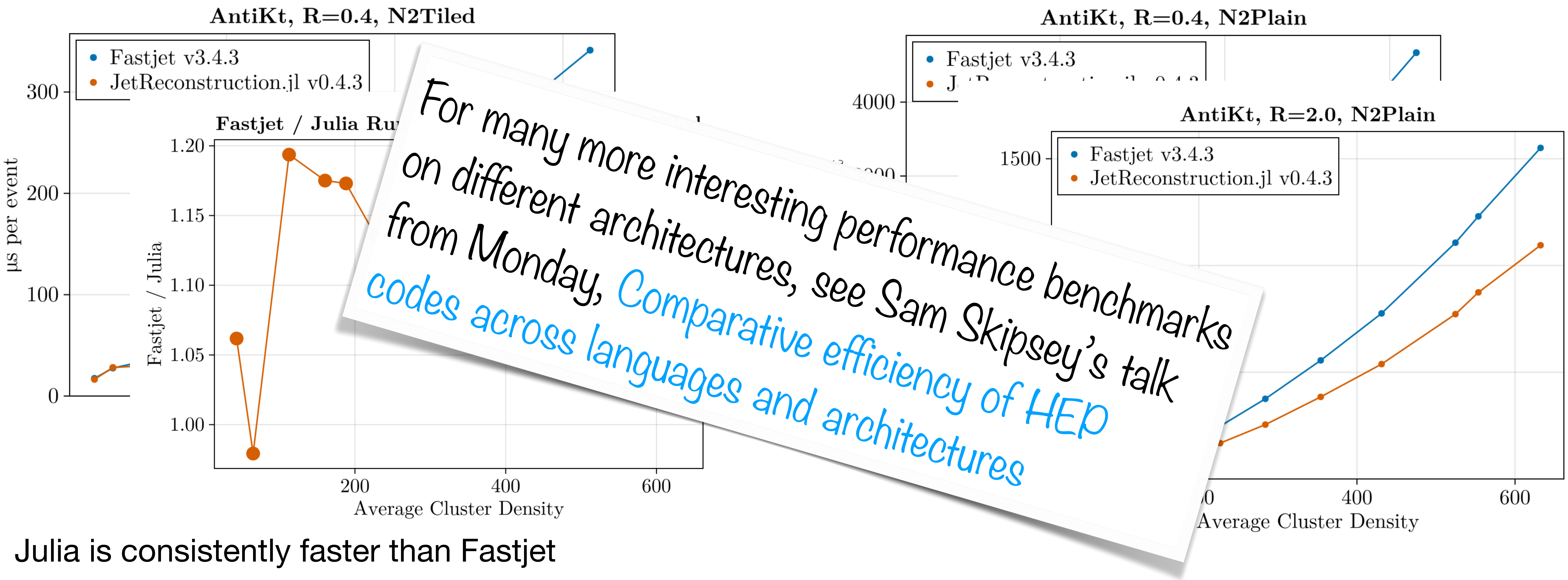




- Julia is consistently faster than Fastjet
  - N2Tiled: Gains are roughly 15% for the tiled algorithm
  - N2Plain: Fastjet for the plain algorithm is a bit pathological at low R - it's better behaved for $R \geq 1.0$
- Julia wins by taking more advantage of SIMD and loop vectorisation

* See backup for benchmark parameters

pp Algorithm Performance*



For many more interesting performance benchmarks on different architectures, see Sam Skipsey's talk from Monday, Comparative efficiency of HEP codes across languages and architectures

- Julia is consistently faster than Fastjet
  - N2Tiled: Gains are roughly 15% for the tiled algorithm
  - N2Plain: Fastjet for the plain algorithm is a bit pathological at low R - it's better behaved for $R \geq 1.0$
- Julia wins by taking more advantage of SIMD and loop vectorisation

* See backup for benchmark parameters

e^+e^- Jet Reconstruction

- The core of the algorithms used is the same as for pp events
  - Find the nearest neighbour *geometrically* for all pairs of pseudo jets
    - i.e., in (θ, ϕ) space, instead of (y, ϕ)
  - Calculate a metric distance, d_{ij}, between these NN pairs
  - For the lowest value of the d_{ij}, merge these two pseudo jets into one
    - Or in some cases finalise that jet (a.k.a. "beam merge")
  - Keep going until all jets are merged or finalised

A Tale of Three Algorithms…

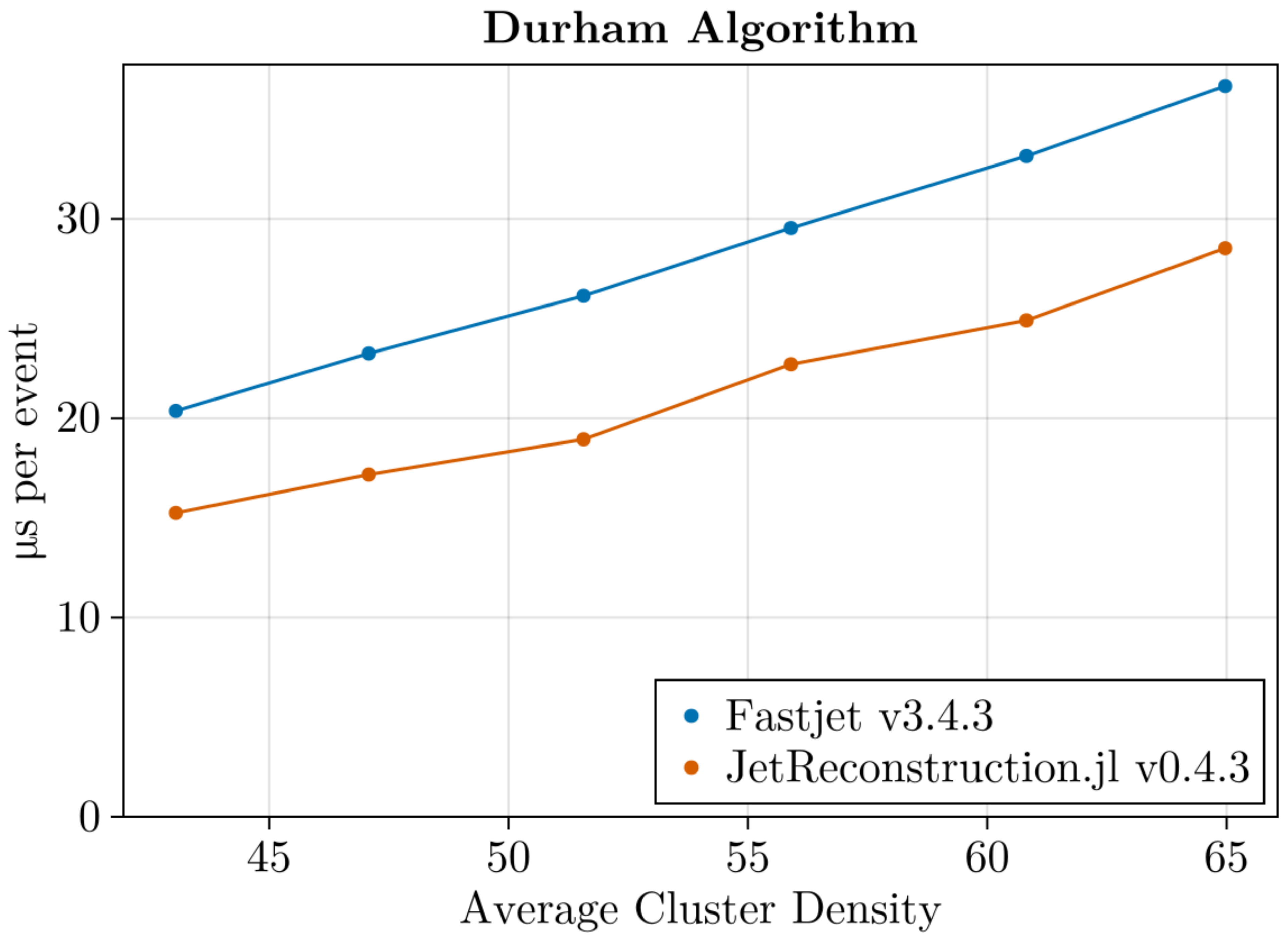
	Durham e^+e^-	Generalised kT e^+e^-	AntiKt pp
Geometric Distance	$1 - \cos\Theta_{ij}$	$1 - \cos\Theta_{ij}$	$\sqrt{\Delta y_{ij}^2 + \Delta\theta_{ij}^2}$
Metric Distance, d_{ij}	$2\min(E_i^2, E_j^2)(1 - \cos\theta_{ij})$	$\min(E_i^{2p}, E_j^{2p})\frac{1 - \cos\theta_{ij}}{1 - \cos R}$	$\min(p_{Ti}^{-2}, p_{Ti}^{-2p})\frac{\sqrt{\Delta y_{ij}^2 + \Delta\theta_{ij}^2}}{R}$
Parameters		p, R	R
Notes		For $p = 1, \pi < R < 3\pi$, equivalent to Durham	If p_T^{2p} then $p = \{-1,0,1\}$ gives {AntiKt, Cambridge/Aachen, Inclusive Kt}

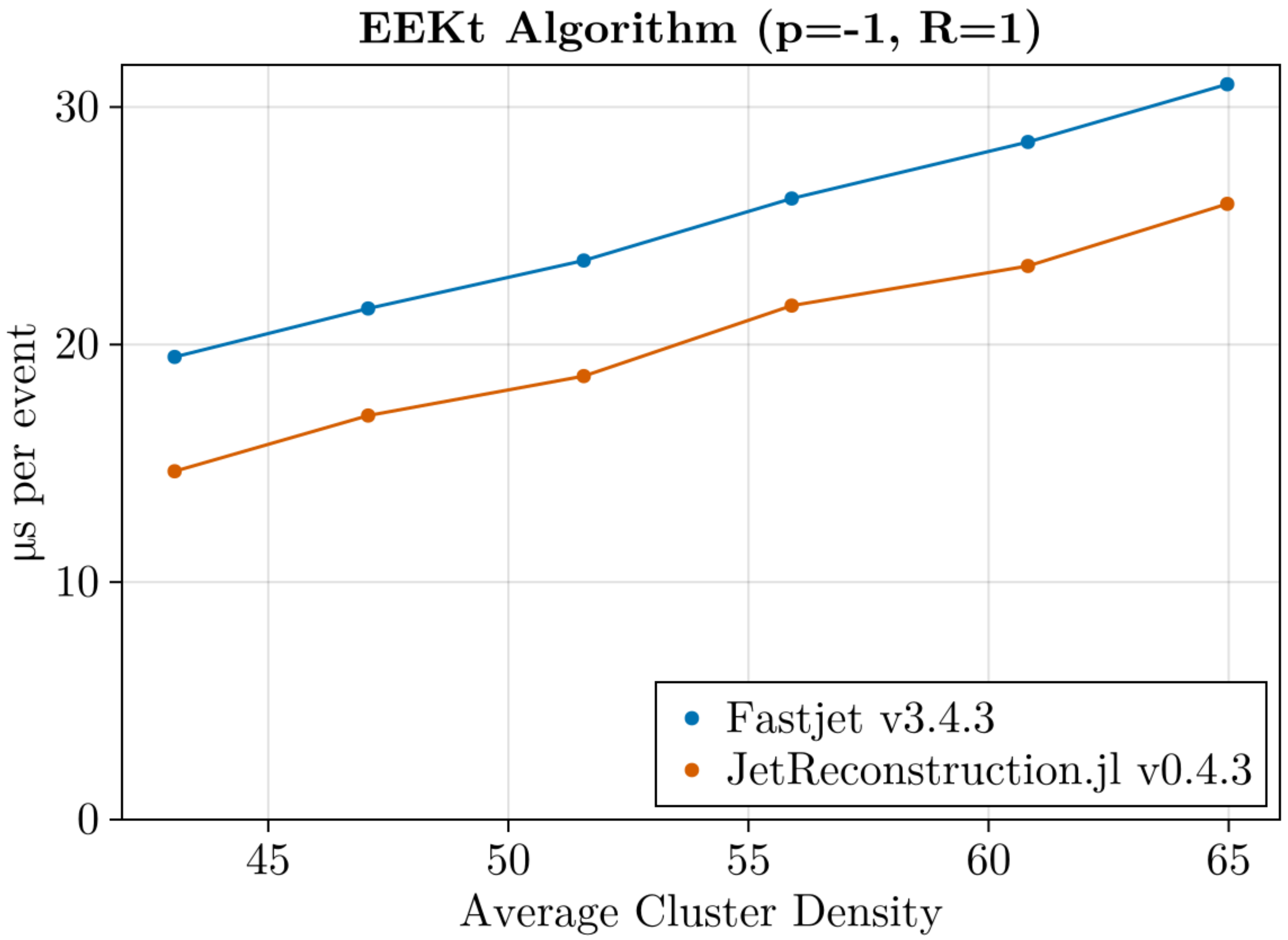
Θ_{ij} is the angle between jets i and j

A Few Implementation Details…

- The `PseudoJet` class used in the pp reconstruction wasn't very suitable for e^+e^-
  - It is working in (y, ϕ) space not (θ, ϕ) space
  - Want to cache normalised momenta to calculate d_{ij} from a dot product
- We introduced a new `EEJet` class
  - Both are concrete subtypes of abstract `FourMomentum` (as is `PseudoJet` for pp)
    - In the pipeline is to update all of these types to `AbstractLorentzVector` as a type we use across the ecosystem
- During the reconstruction an optimised *Structure of Arrays* layout is used
  - This is beautifully easy to do, thanks to Julia's `StructArrays.jl` package
- The tiled strategy is *not* implemented here
  - Particle densities are too low to make this worthwhile
- Released as v0.4.0 (Durham) and v0.4.1 (Generalised e^+e^- Kt)
  - Accidental performance regression - fixed in v0.4.3!

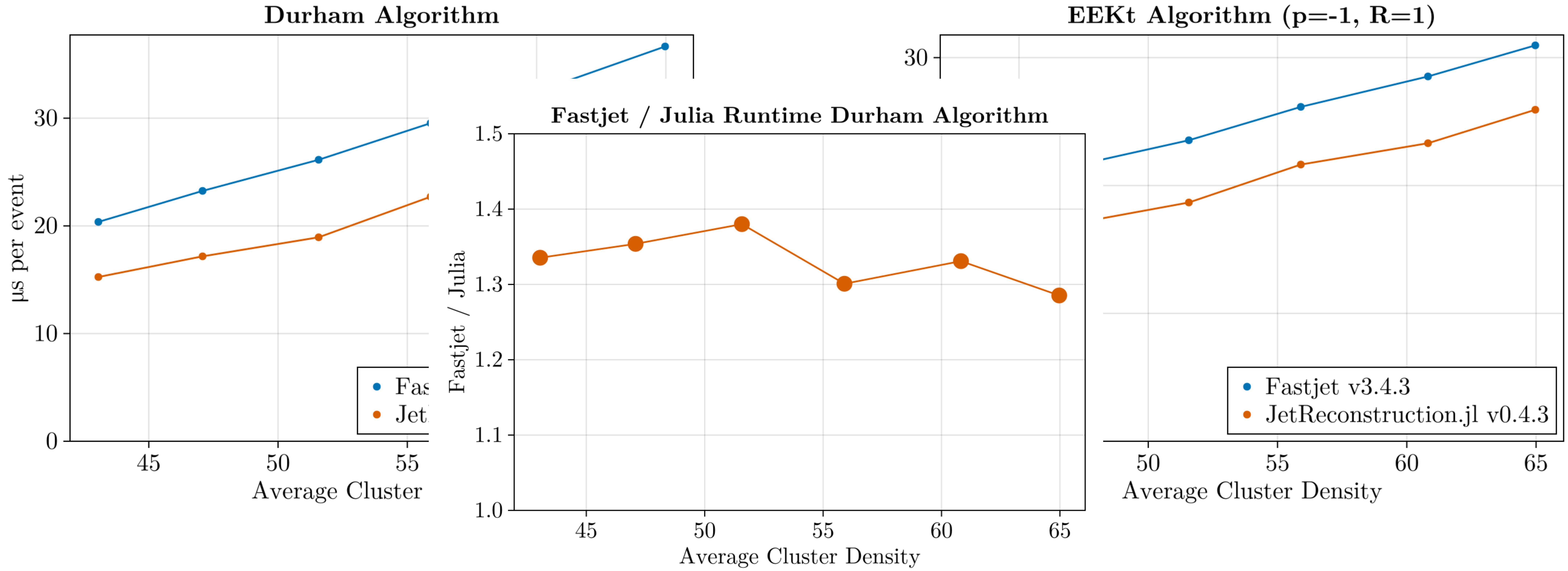
e+e- Algorithm Performance





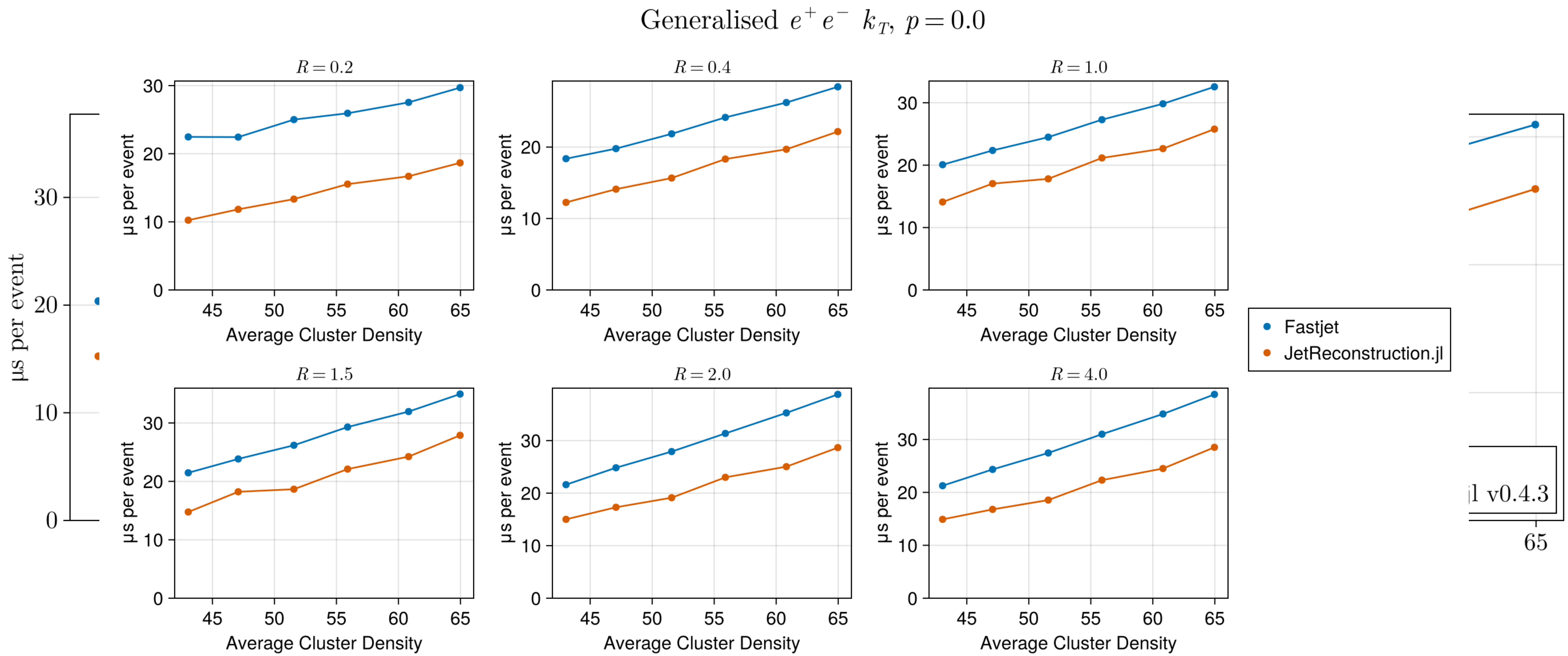
- Consistently faster than Fastjet by ~30% for Durham algorithm
- Generalised e^+e^- similarly better than Fastjet by ~20% at most R values

e^+e^- Algorithm Performance



- Consistently faster than Fastjet by ~30% for Durham algorithm
- Generalised e^+e^- similarly better than Fastjet by ~20% at most R values

e+e- Algorithm Performance



- Consistently faster than Fastjet by ~30% for Durham algorithm
- Generalised e^+e^- similarly better than Fastjet by ~20% at most R values

FCCee Jets!

- All test up to now have used ASCII HepMC3 files
  - Read and converted into suitable internal EDM types
  - A bit tedious for the user - we want to read their EDM directly
- For future collider studies we can!
  - Take advantage of UnROOT.jl to read EDM4hep files
  - And EDM4hep.jl to interpret the data into a nice data model in Julia
- What's needed?
  - Define the converters from EDM4hep Reconstructed particles into JetReconstruction's EDM

Both packages presented at this CHEP

FCCee Jets!

Implemented as a package extension - only loaded when EDM4hep is used

```
JetReconstruction.px(rp::ReconstructedParticle) = rp.momentum.x
JetReconstruction.py(rp::ReconstructedParticle) = rp.momentum.y
JetReconstruction.pz(rp::ReconstructedParticle) = rp.momentum.z
JetReconstruction.energy(rp::ReconstructedParticle) = rp.energy

function JetReconstruction.EEjet(rp::ReconstructedParticle)
    EEjet(JetReconstruction.px(rp), JetReconstruction.py(rp),
          JetReconstruction.pz(rp), JetReconstruction.energy(rp))
end
```

- All test up to now have used ASCII HepMC3 files
  - Read and converted into suitable internal EDM types
  - A bit tedious for the user - we want to read their EDM directly
- For future collider studies we can!
  - Take advantage of UnROOT.jl to read EDM4hep files
  - And EDM4hep.jl to interpret the data into a nice data model in Julia
- What's needed?
  - Define the converters from EDM4hep Reconstructed particles into JetReconstruction's EDM

Both packages presented at this CHEP

FCCee Jets!

Implemented as a package extension - only loaded when EDM4hep is used

```
JetReconstruction.px(rp::ReconstructedParticle) = rp.momentum.x
JetReconstruction.py(rp::ReconstructedParticle) = rp.momentum.y
JetReconstruction.pz(rp::ReconstructedParticle) = rp.momentum.z
JetReconstruction.energy(rp::ReconstructedParticle) = rp.energy

function JetReconstruction.EEjet(rp::ReconstructedParticle)
    EEjet(JetReconstruction.px(rp), JetReconstruction.py(rp),
          JetReconstruction.pz(rp), JetReconstruction.energy(rp))
end
```

- All test up to now have used ASCII HepMC3 files
  - Read and converted into suitable internal EDM types
  - A bit tedious for the user - we want to read their EDM directly
- For future collider studies we can!
  - Take advantage of UnROOT.jl to read EDM4hep files
  - And EDM4hep.jl to interpret the data into a nice data model in Julia
- What's needed?
  - Define the converters from EDM4hep Reconstructed particles into JetReconstruction's EDM

Both packages presented at this CHEP

```
input_file = joinpath("events_196755633.root")
reader = RootIO.Reader(input_file)
events = RootIO.get(reader, "events")

evt = events[1]

recps = RootIO.get(reader, evt, "ReconstructedParticles")

cs = jet_reconstruct(recps; algorithm = JetAlgorithm.Durham)
for jet in exclusive_jets(cs; njets = 2, T = EEjet)
    println(jet)
end
```

FCCee Jets!

Implemented as a package extension - only loaded when EDM4hep is used

```julia
JetReconstruction.px(rp::ReconstructedParticle) = rp.momentum.x
JetReconstruction.py(rp::ReconstructedParticle) = rp.momentum.y
JetReconstruction.pz(rp::ReconstructedParticle) = rp.momentum.z
JetReconstruction.energy(rp::ReconstructedParticle) = rp.energy

function JetReconstruction.EEjet(rp::ReconstructedParticle)
    EEjet(JetReconstruction.px(rp), JetReconstruction.py(rp),
          JetReconstruction.pz(rp), JetReconstruction.energy(rp))
end
```

- All test up to now have used ASCII HepMC3 files
  - Read and converted into suitable internal EDM types
  - A bit tedious for the user - we want to read their EDM directly
- For future collider studies we can!
  - Take advantage of UnROOT.jl to read EDM4hep files
  - And EDM4hep.jl to interpret the data into a nice data model in Julia
- What's needed?
  - Define the converters from EDM4hep Reconstructed particles into JetReconstruction's EDM

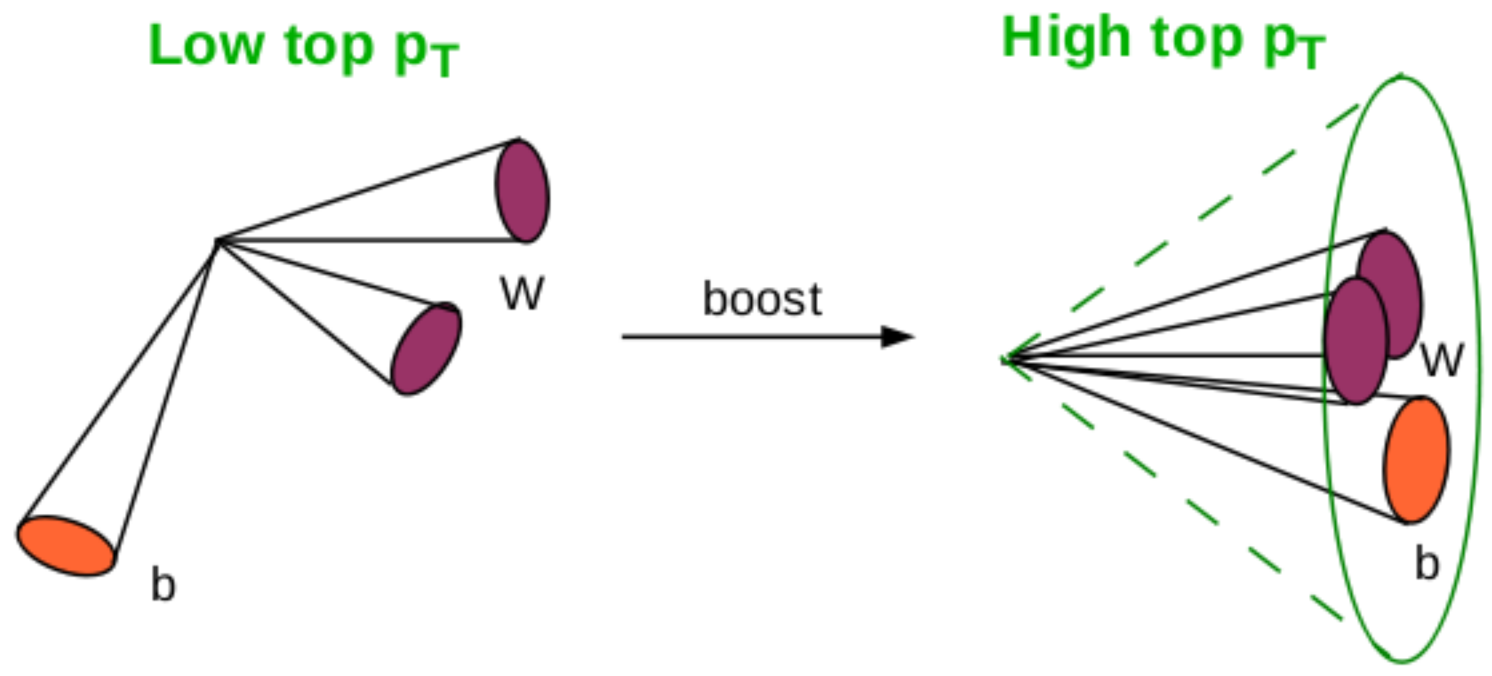
Both packages presented at this CHEP

```julia
input_file = joinpath("events_196755633.root")
reader = RootIO.Reader(input_file)
events = RootIO.get(reader, "events")

evt = events[1]

recps = RootIO.get(reader, evt, "ReconstructedParticles")

cs = jet_reconstruct(recps; algorithm = JetAlgorithm.Durham)
for jet in exclusive_jets(cs; njets = 2, T = EEjet)
    println(jet)
end
```

On a single thread we can process EDM4hep jets at 24kHz (jet reco only)

Substructure and Taggers

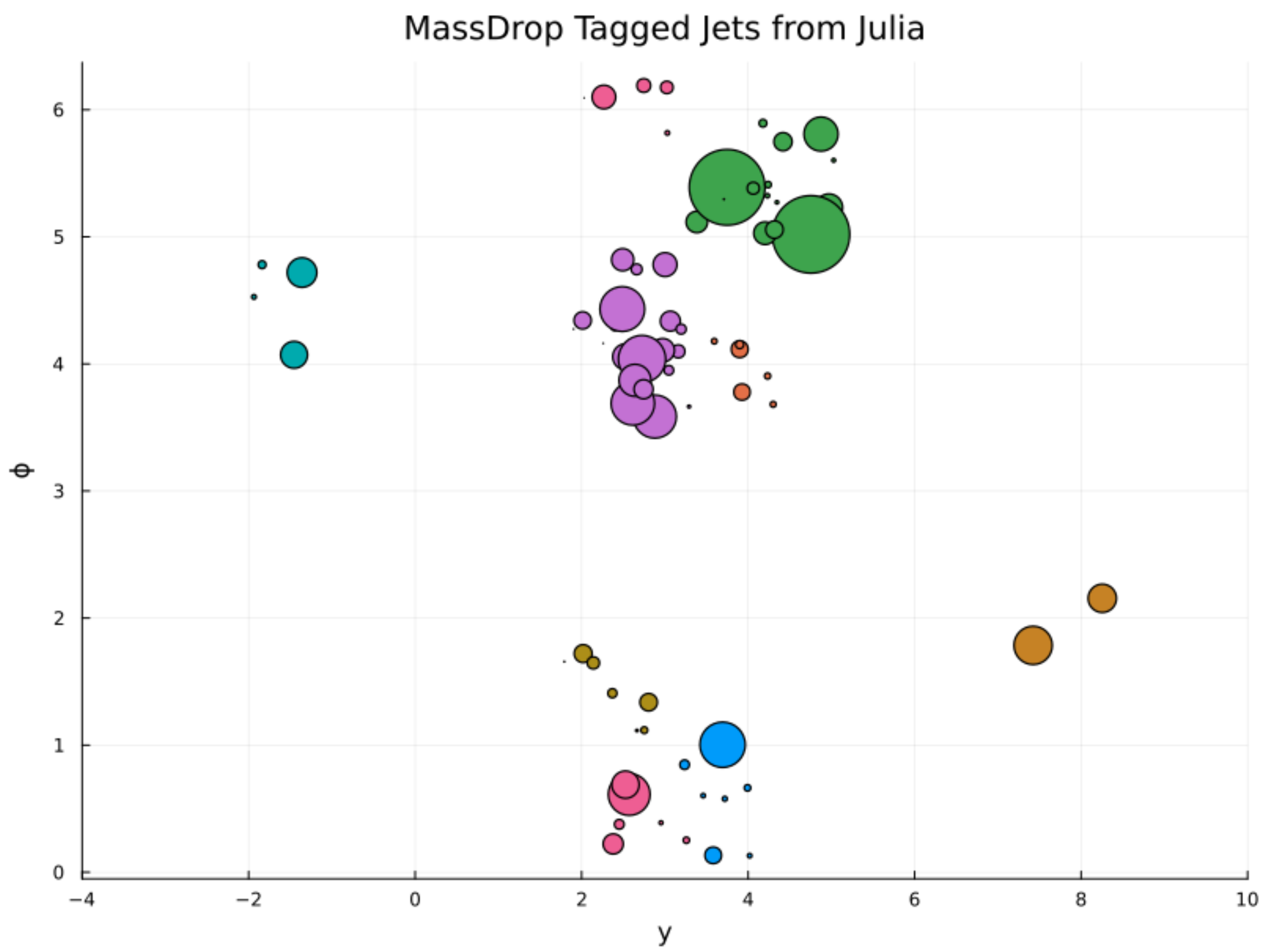
- Finding jets and looking at exclusive and inclusive samples is on the beginning
  - Substructure provides more information about the particles which initiated the jet (q, g, W, Z, H, …)
  - Essential for identifying boosted heavy particles as part of the search for new physics

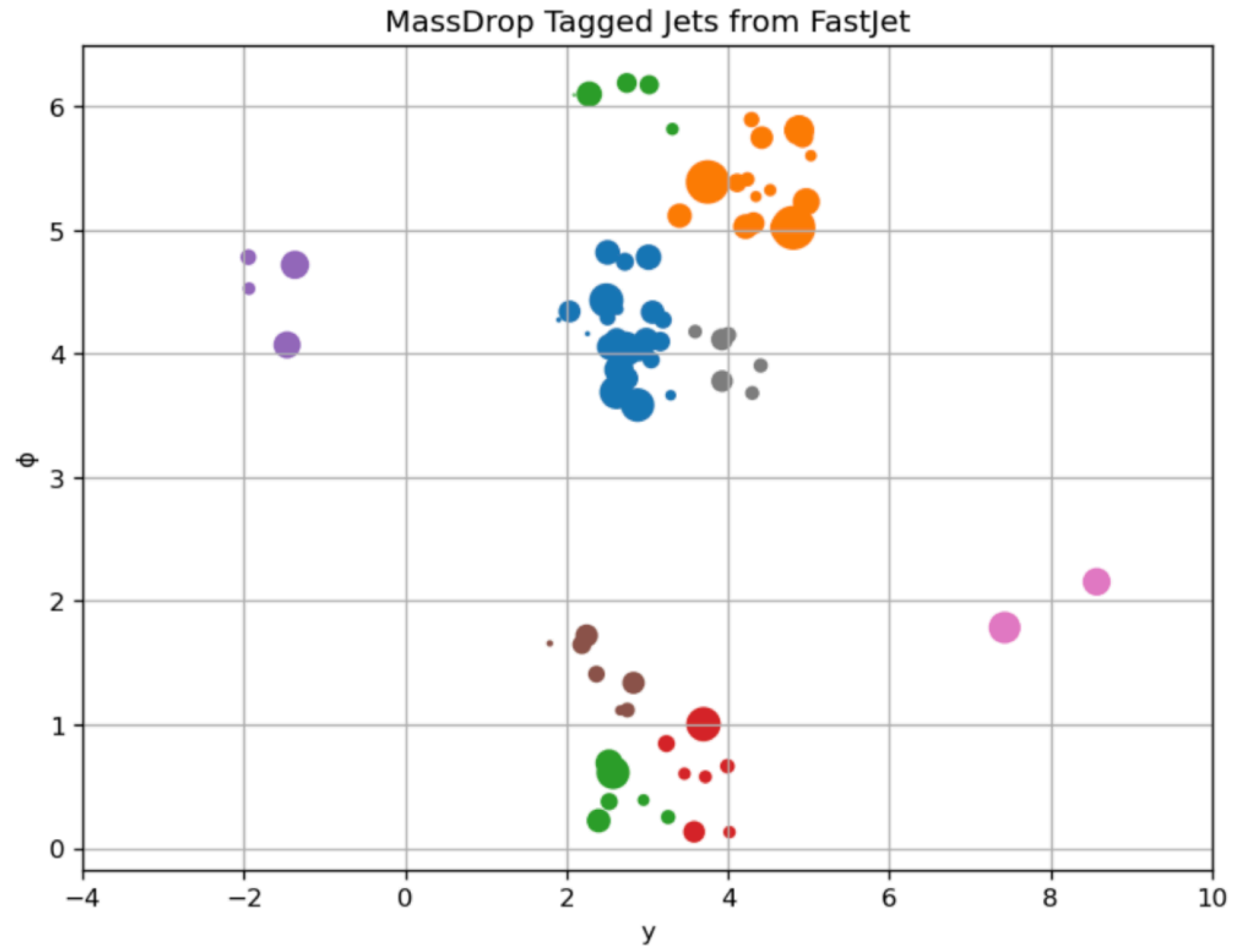


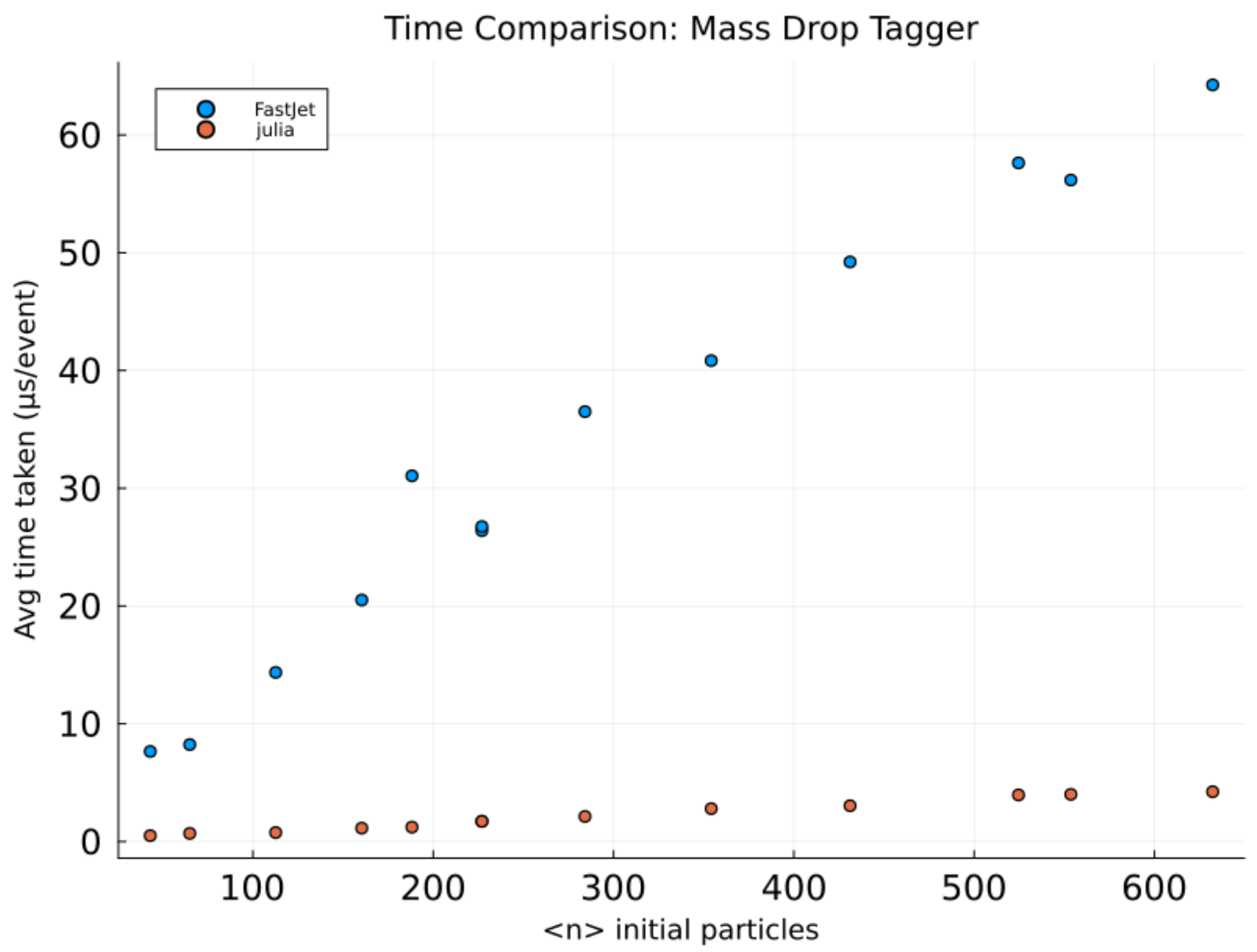
Mass Drop Tagger

Also implemented: the John Hopkins tagger

- Walk back through the clustering sequence of Cambridge/Aachen, splitting a jet j into its two ancestors j_1 and j_2
  - If there is a signifiant mass drop, $m_{j1} < \mu m_j$ then j is tagged (subject to an asymmetry criterion)
  - Otherwise, repeat using j_1
- Same results as Fastjet, much better runtime



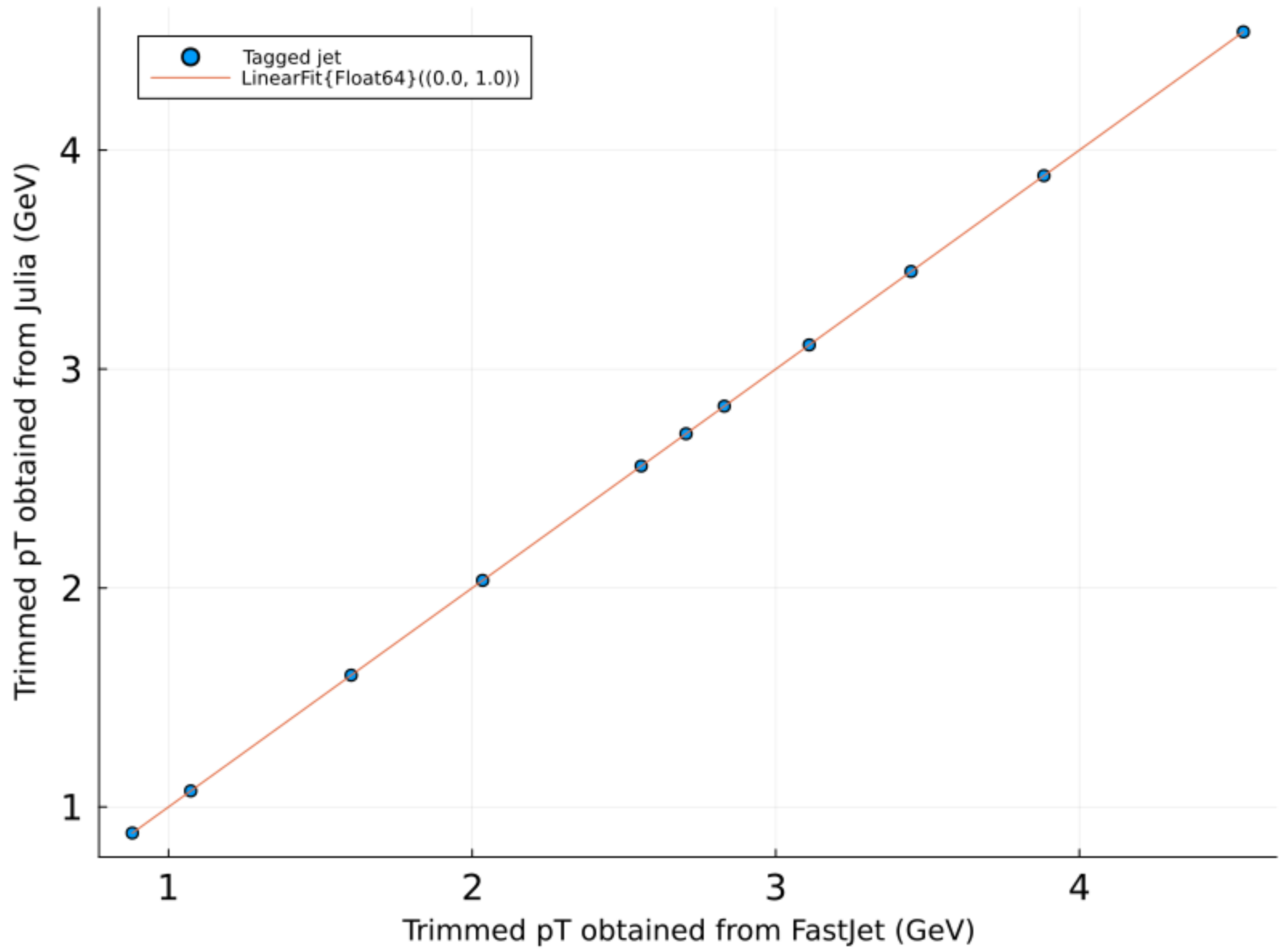


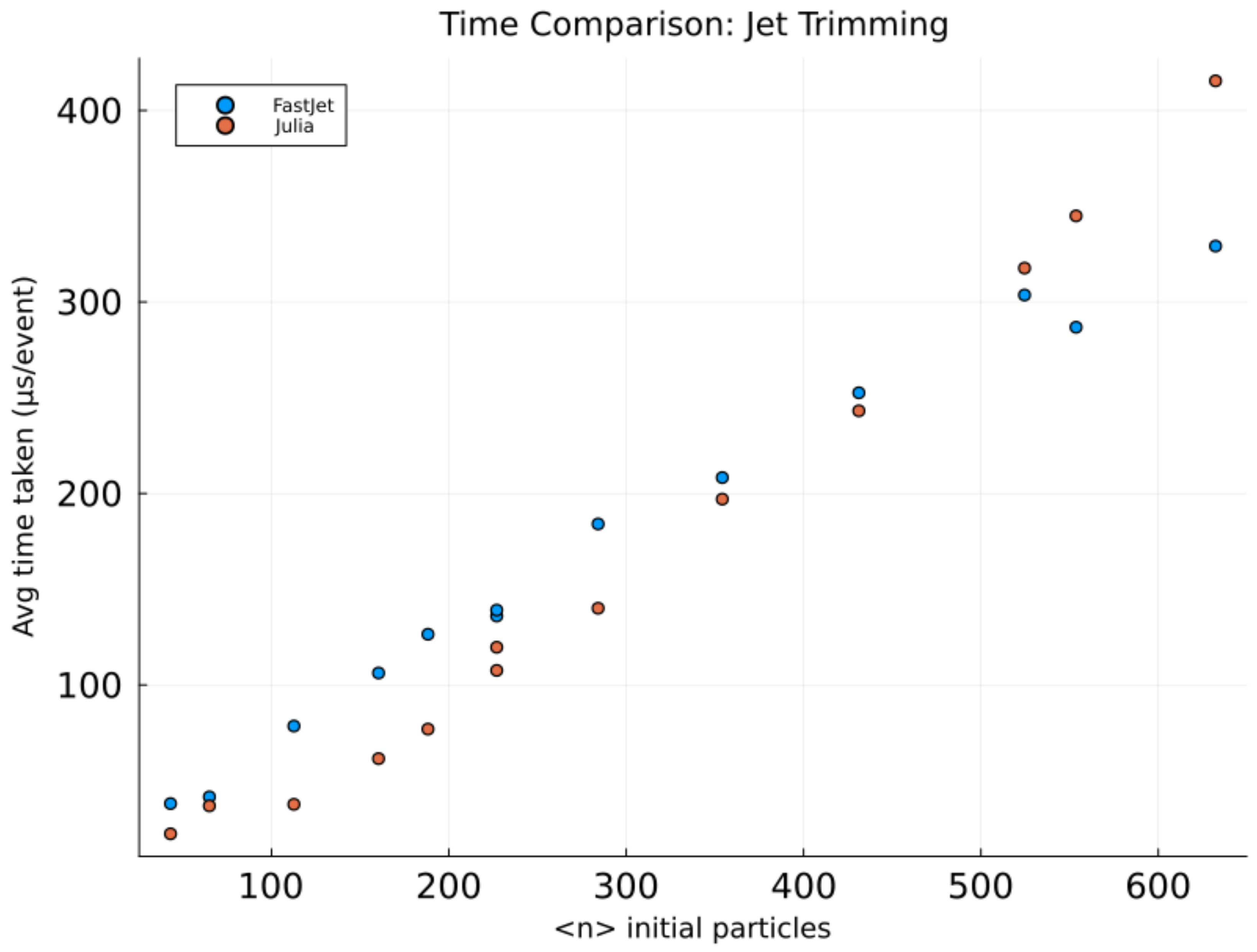


Jet Trimming

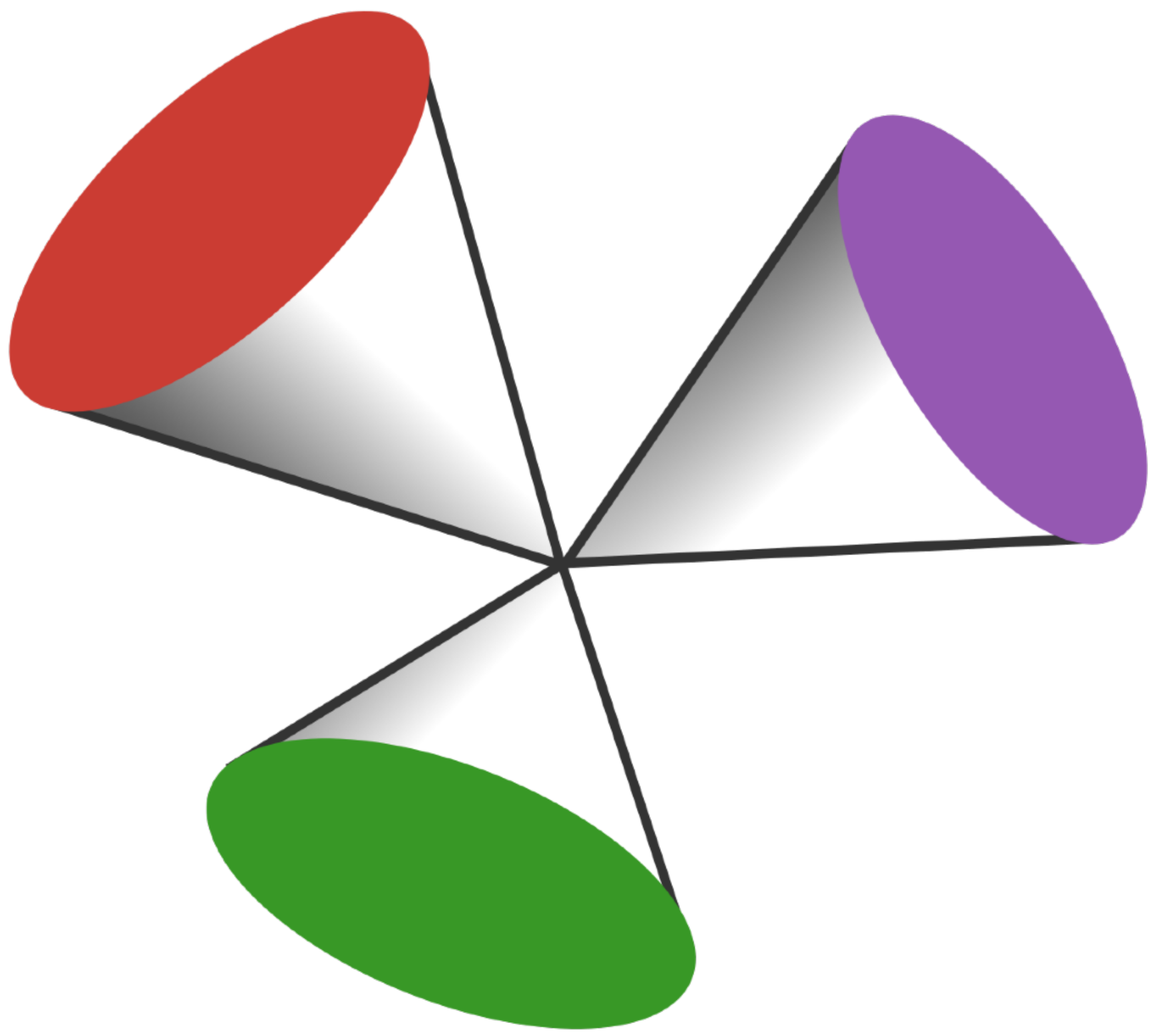
Also implemented: jet filtering

- Trimming removes a jet's soft components
  - Ideally cutting out spurious pile-up radiation
- Same results as Fastjet, about the same runtime





Status and Outlook



- JetReconstruction.jl is available: easy to use, works really well!
  - Growing set of features
    - Most important/popular algorithms for pp and e^+e^-
    - Jet selections
    - Tagging, trimming and filtering (to be merged)
    - Direct EDM imports - should be easy to add more
      - Will be boosted by `AbstractLorentzVector` work from JuliaHEP community
  - Largely faster than Fastjet by 15-30% for realistic cases 🎉
  - Seen use in ATLAS, CMS and in FCCee
    - Improved jet constituent interfaces in the pipeline, discussing with users
- Builds on the very positive experience of using Julia for performance and ergonomics

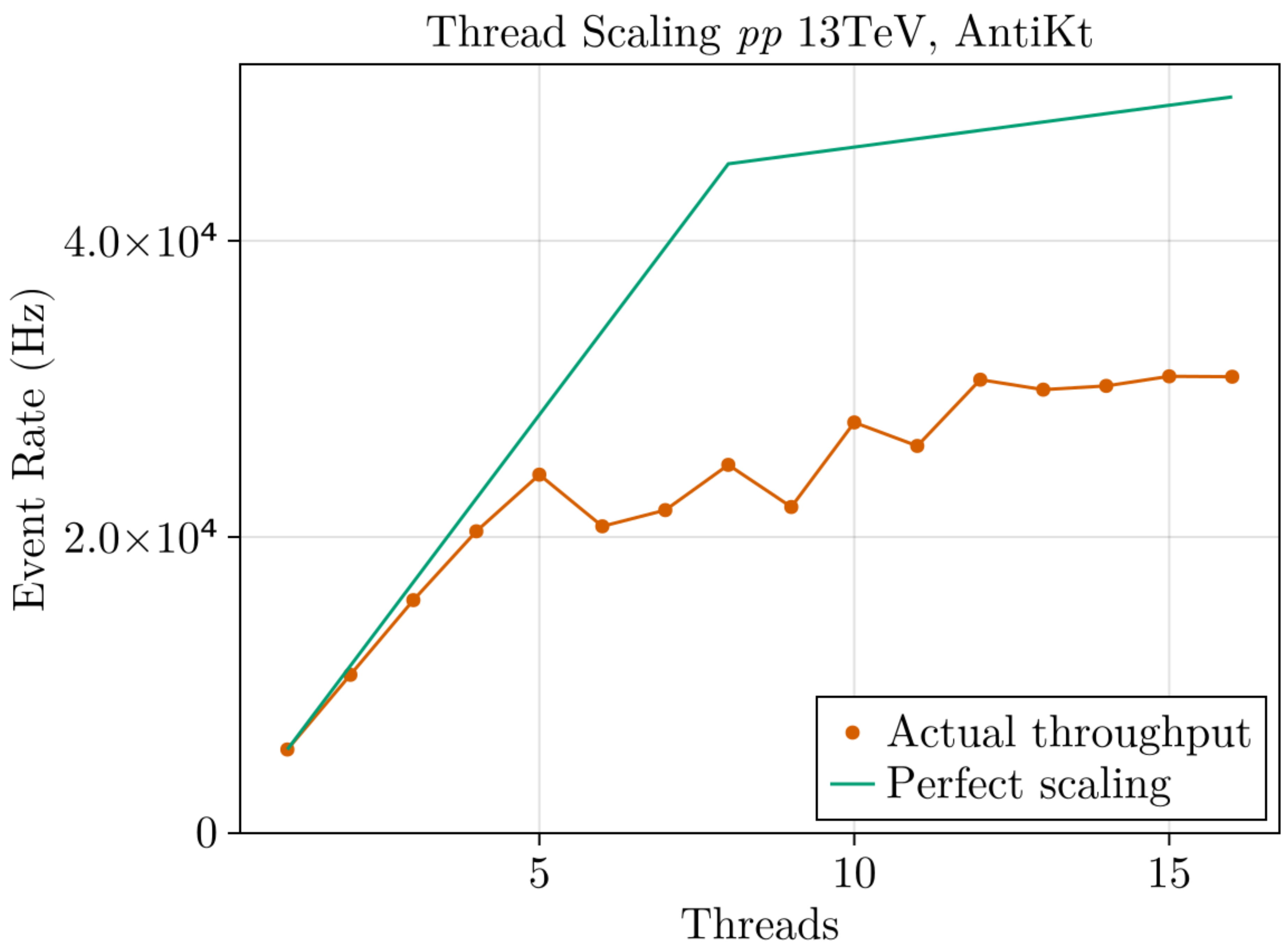
Please try it out, tell us your wishes, give us your feedback!

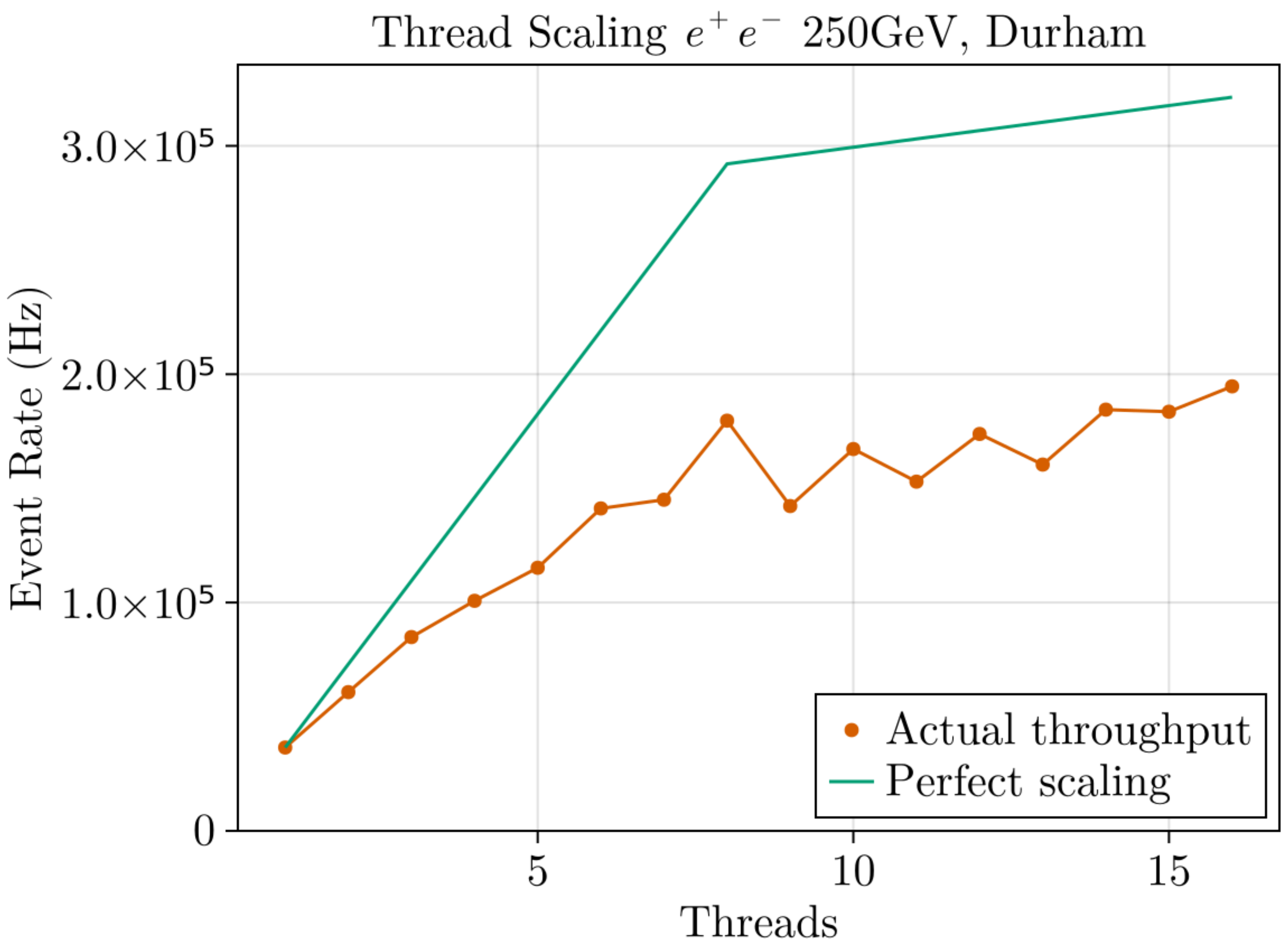
Backup

Benchmark Parameters

- JetReconstruction: AMD Ryzen 7, 5700G 3.8GHz (8 cores, plus HT), 32GB RAM, AlmaLinux 9.4
- JetSubstructure: M1, OSX 14.5
- Codes:
  - Julia v1.11.1; JetReconstruction v0.4.3; julia-JetSubstructure
    - Benchmark helpers in JetReconstructionBenchmarks.jl (including Pythia generated source events)
  - Fastjet v3.4.3 (compiled with gcc 11.4.1, -O2)
- Jitter - benchmark values taken over 32 runs and are stable to ~1%
- More platforms:
  - See Sam Skipsey's talk, Comparative efficiency of HEP codes across languages and architectures

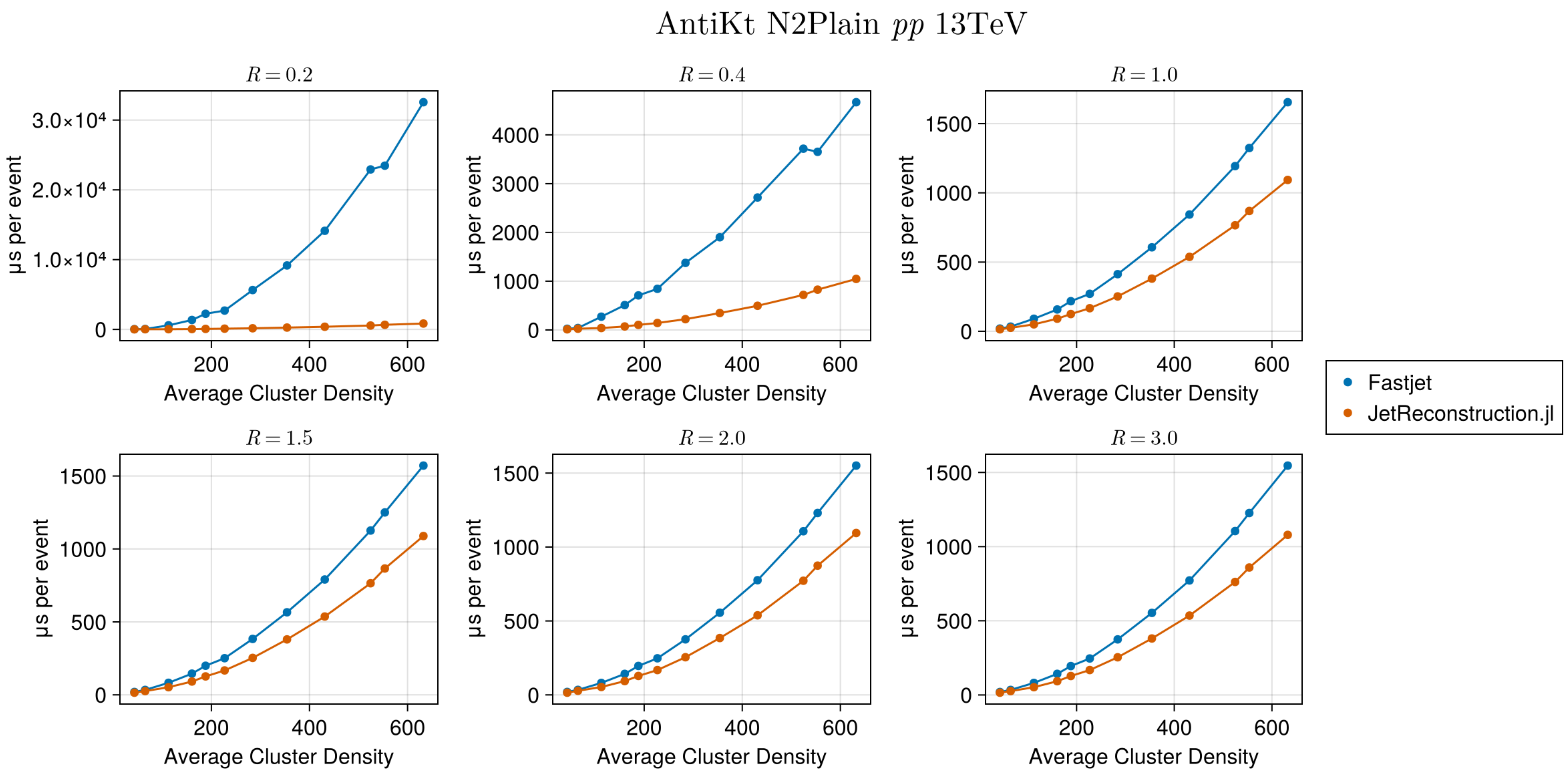
Multi-threaded Performance



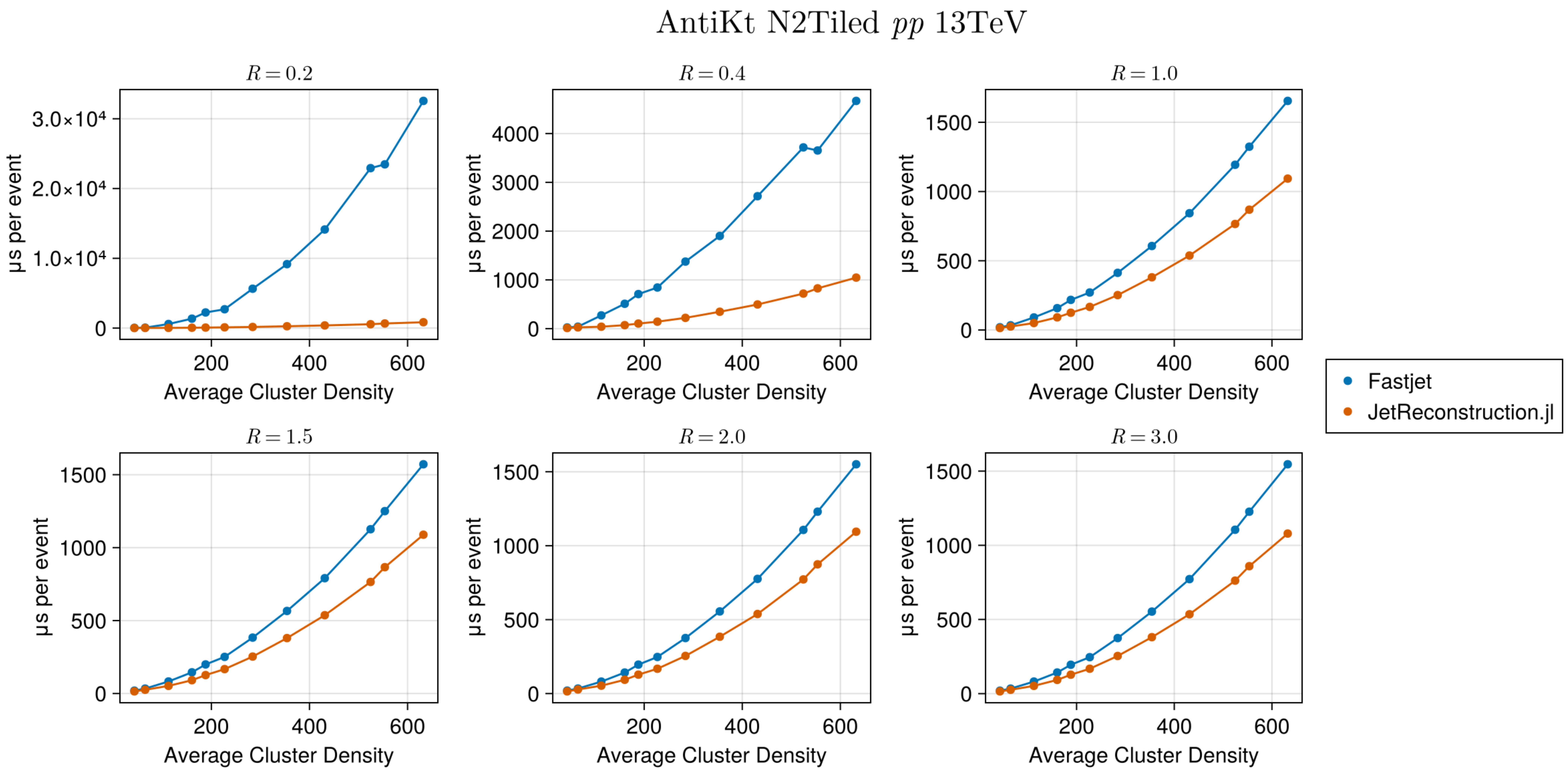


- Simple multithreading activated by `@threads`
  - Good: it works!
  - Bad: performance a bit lacklustre after ~5 threads

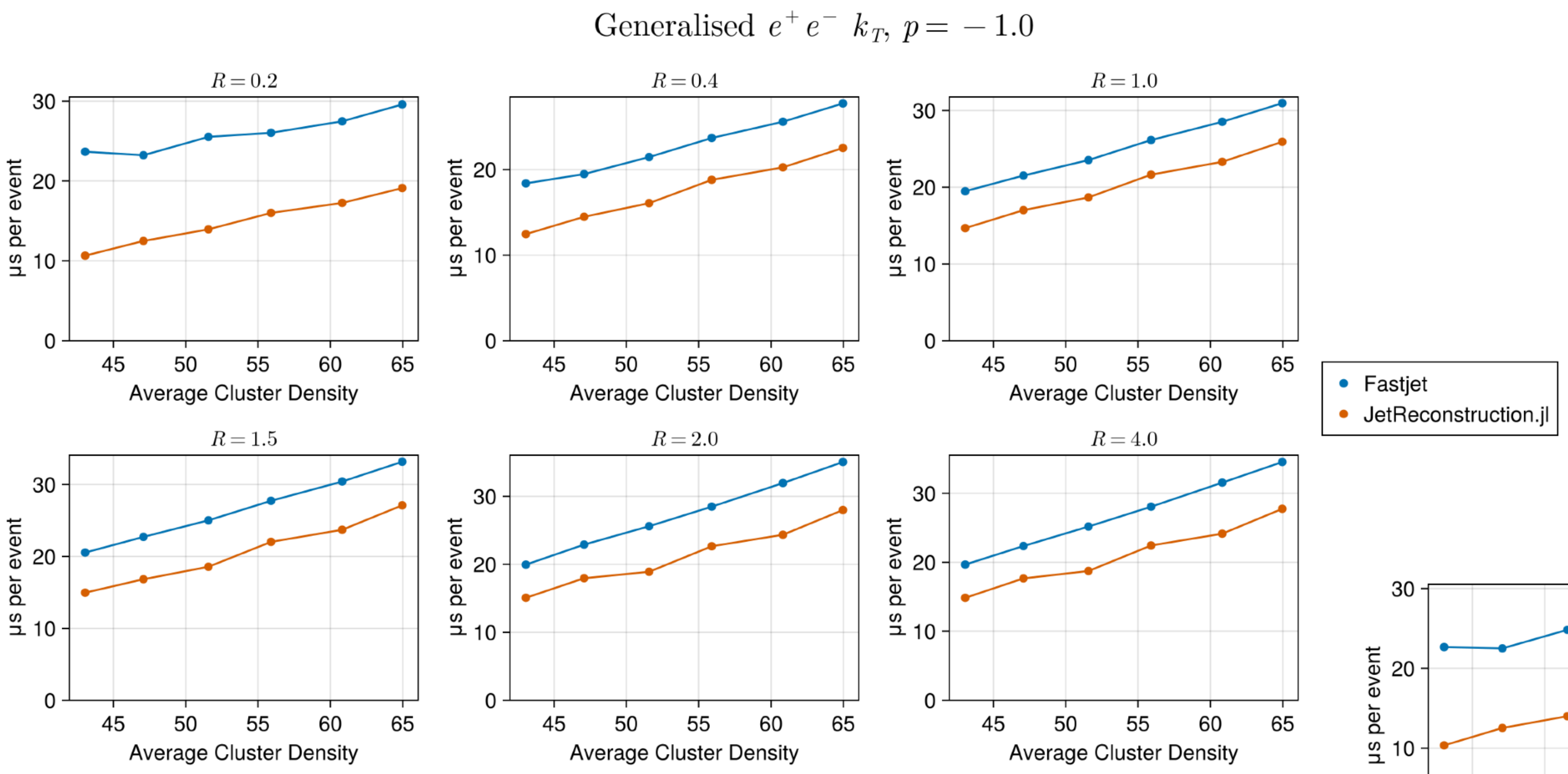
Variation with R: N2Plain

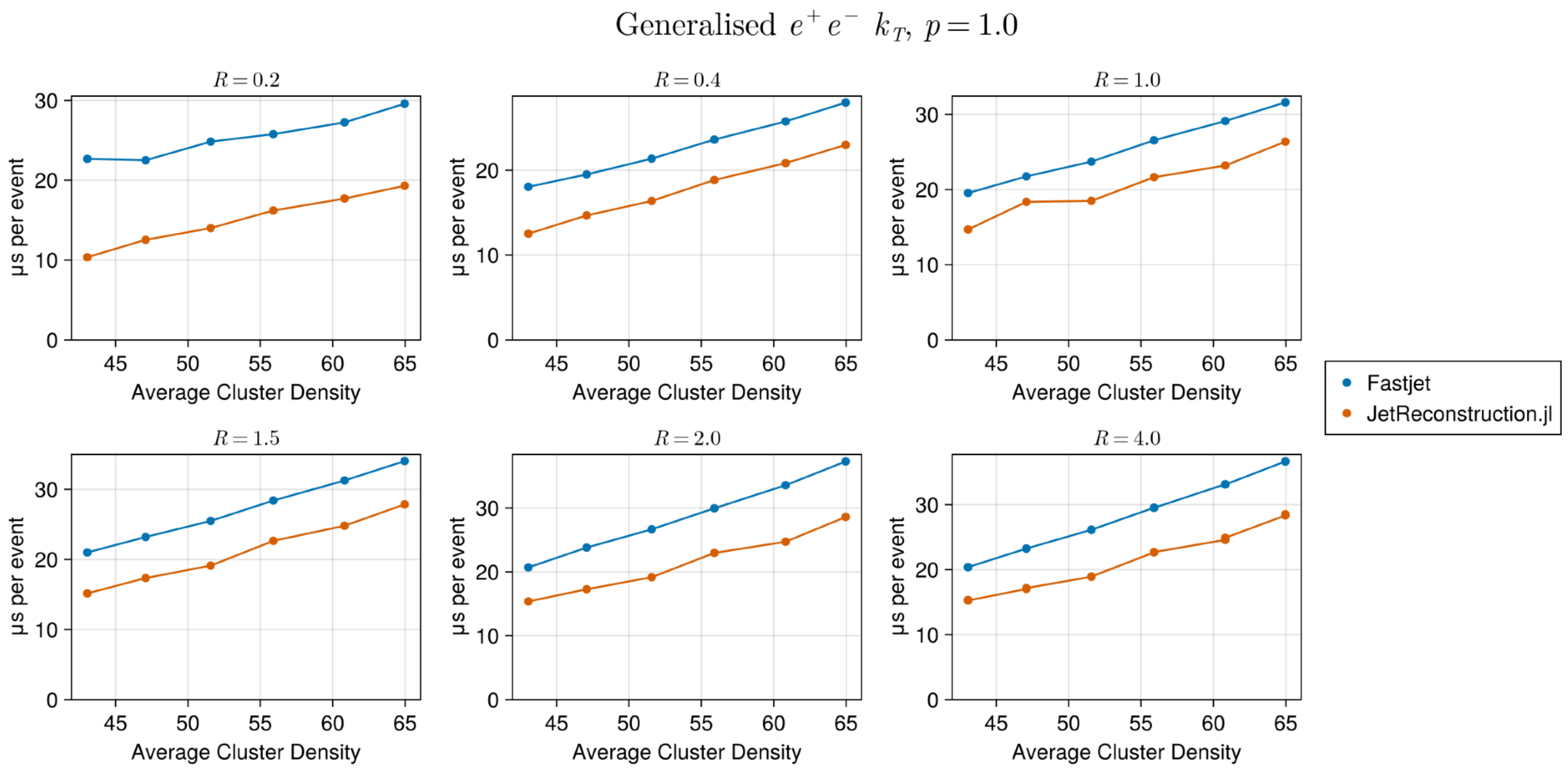


Variation with R: N2Tiled

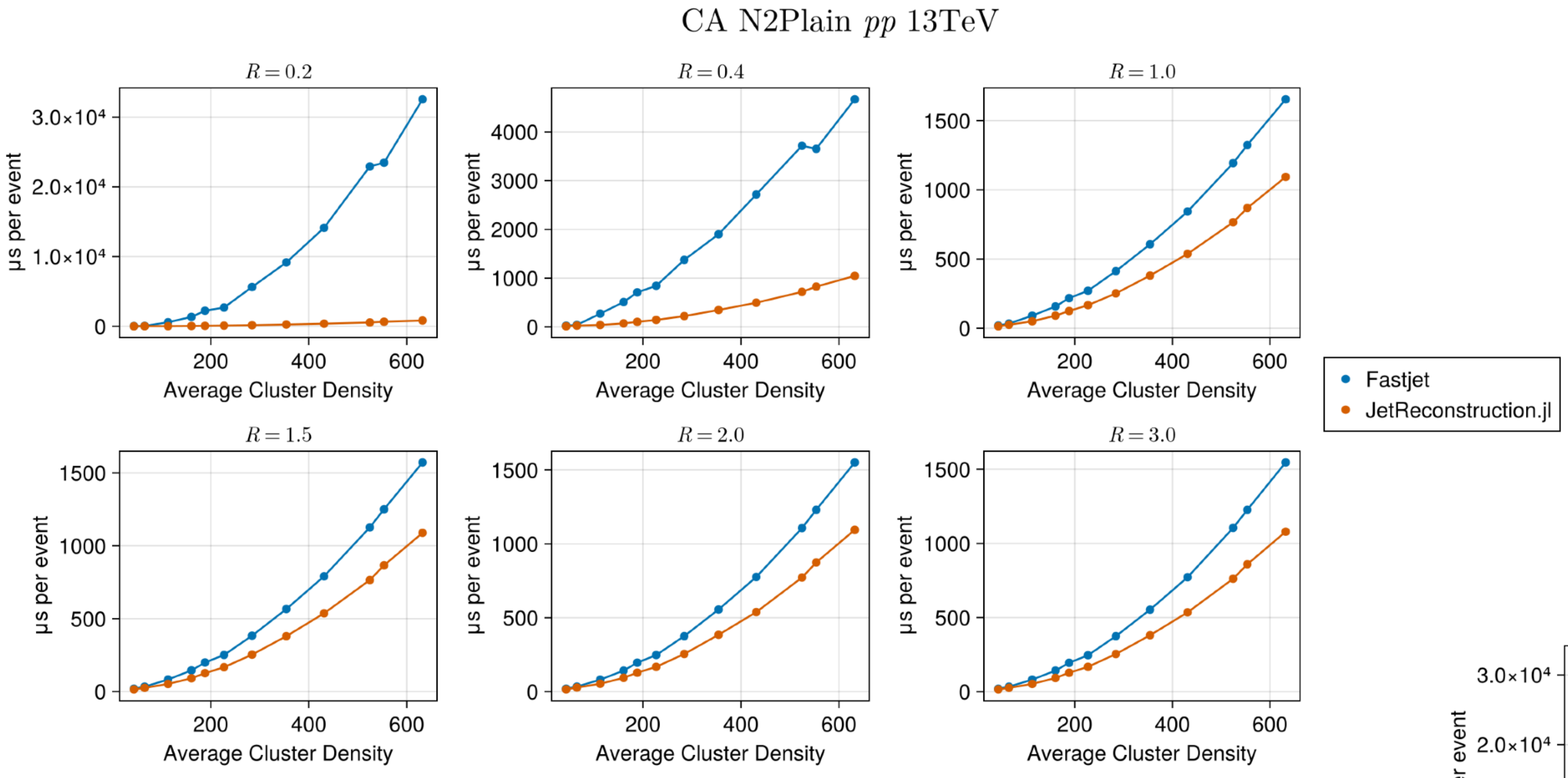


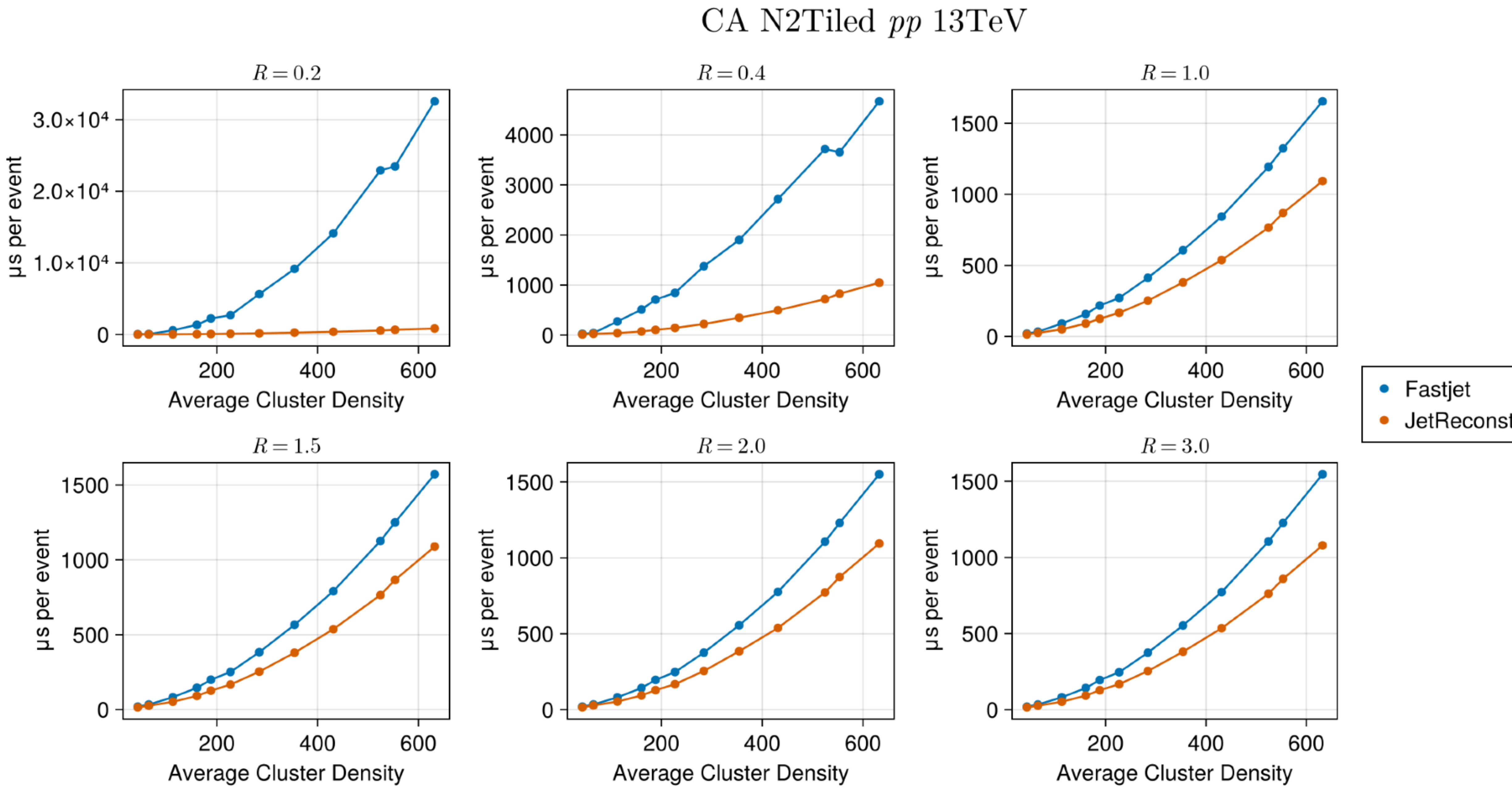
Generalised e^+e^- Kt





Cambridge/Aachen for *pp*





Inclusive Kt for *pp*