Jet Finding in Julia

JuliaHEP Workshop 2023

Graeme Stewart, Philippe Gras, Atell Krasnopolski

CERN

cea irfu

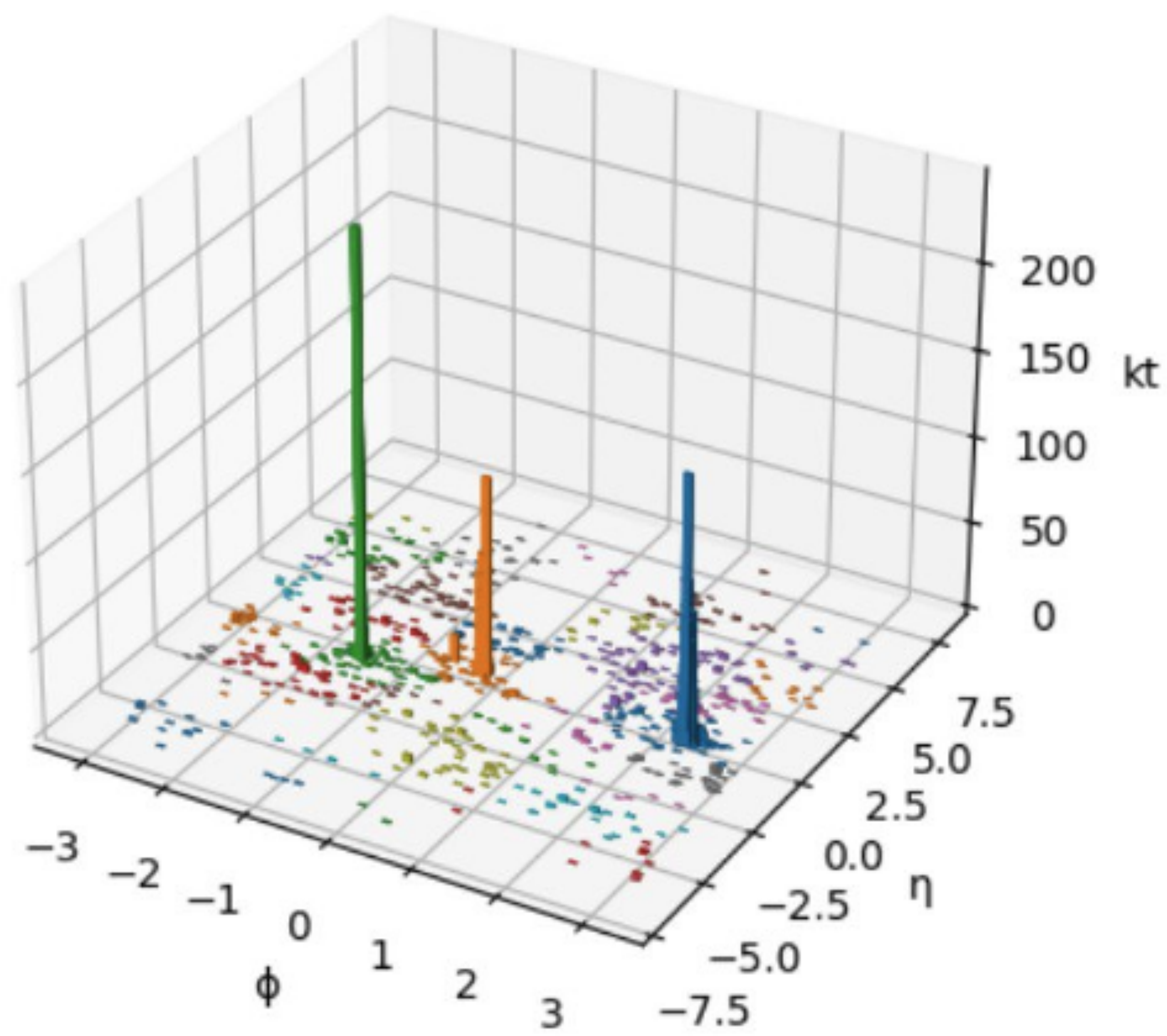
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

HEP and Programming Languages

- Languages in HEP do evolve - albeit slowly!
  - Originally we programmed in Fortran for LEP (cf. Jim's talk on Monday)
  - With the LHC a wholesale transition to C++ occurred
  - Then supplemented by the addition of Python in specific areas
    - Configuration and steering
    - Analysis codes
    - Machine learning
  - However, importantly backed by performant C++ code underneath
- Evaluation of any new language is multi-dimensional
  - We wanted to look at some aspects of *algorithmic performance* and comment on *language ergonomics* for different language implementations on a non-trivial problem in HEP

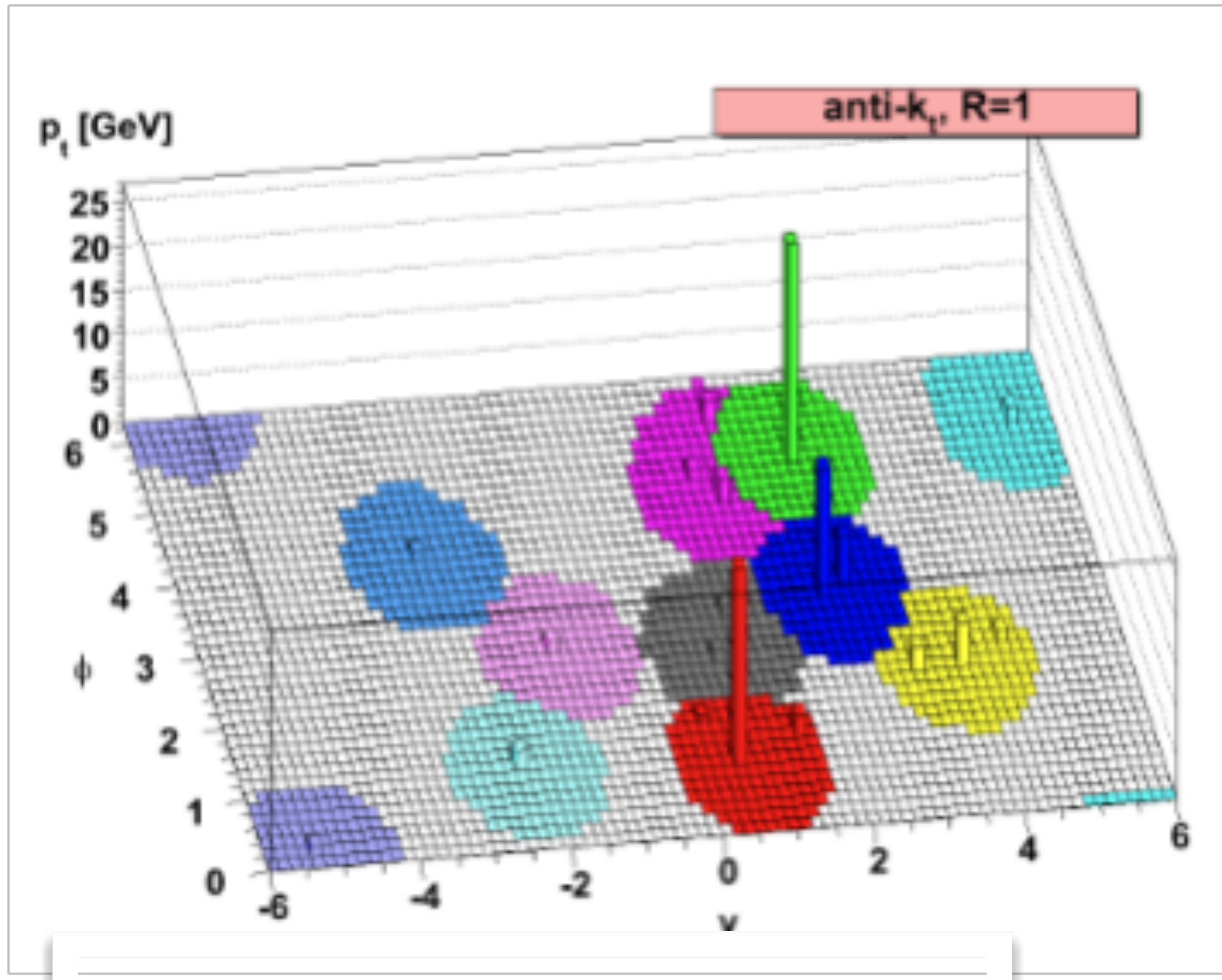
Jet Finding as a Test Case

- Find a non-trivial HEP algorithm
  - Should not be so simple as to add little information over general metrics
  - Should not be so complex that implementation takes a very long time
- Jet finding is a good example of a "goldilocks" algorithm
- The goal is to cluster calorimeter energy deposits into jets
- AntiKt clustering, used by FastJet, is popularly used because it is an infrared and co-linear safe [arXiv:0802.1189]



The Algorithm in Brief

1. Define a distance parameter `R` (we use 0.4, which is 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
        1. $d_{ij} = d \cdot \min(Jet_i\ pt^{2p},\ Jet_j\ pt^{2p})$
        2. N.B. this favours merges with high pt jets, giving stability against soft radiation
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)

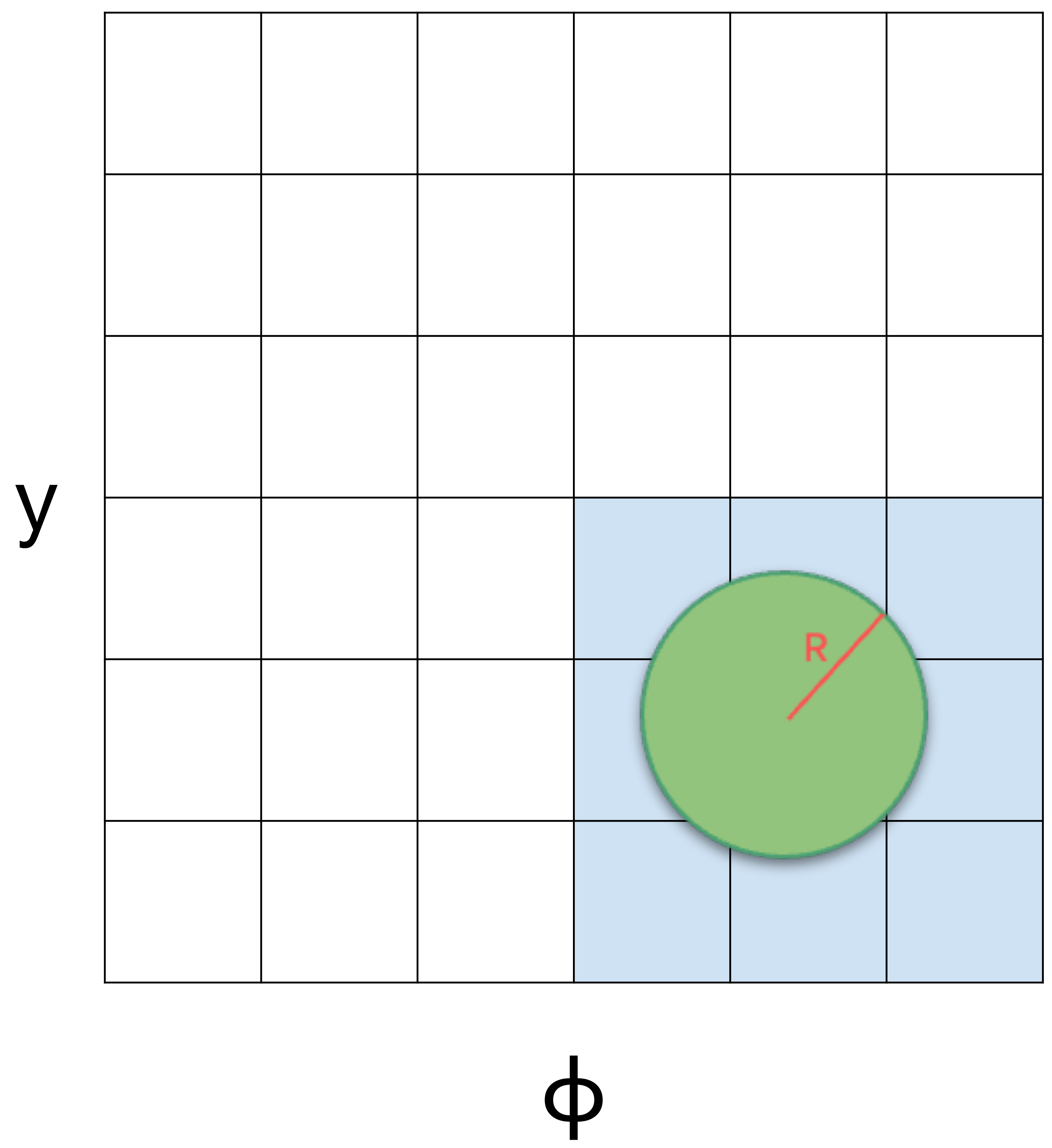
This algorithm due to FastJet [arXiv:1111.6097]

Different Strategies

- We look at two strategies for implementing this algorithm
  - **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 (rapidity, phi) 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



Implementations

- There is a benchmark C++ implementation, used almost ubiquitously in HEP, FastJet (who originally developed the algorithm)
- We initially developed new implementations in Python and Julia
  - With the Python code in two flavours: pure Python and accelerated Python (using numpy and numba)
- Presented initial results at the CHEP2023 conference

Initial Results

- Standard sample 100 of Pythia8 events pp 13TeV, jet pt>20GeV, multiple trials
- Benchmark is C++ N2Tiled strategy at 324μs/event (1.00)
  - All benchmarks repeated multiple times, jitter is < 1%
  - Event read time and also jit time for Numba and Julia is excluded

Implementation	**N2Plain**	**N2Tiled**
C++ (FastJet)	17.6	**1.00**
Python (Pure)	966	222
Python (Accelerated)	53.4	178
Julia	**4.00**	1.12

- Python implementations were really not competitive, so we didn't try to further improve them
  - We also found that the ergonomics of the accelerated Python is suboptimal cf. pure Python or Julia
- The impressive result of the N2Plain algorithm in Julia can be attributed to an SoA layout and SIMD optimisations

Ergonomics Experience

- C++ FastJet code is actually very like C
  - Well written, but some definite tricky parts (pointers to pointers)
- Python
  - Pure Python code is rather easy to use and reason about
  - Numba/numpy accelerated code becomes unweildy as the problem needs to be cast into a numpy array layout
    - Also numba acceleration doesn't work for quite a few things
- Julia
  - As easy as pure Python for the basic implementation parts
    - Particularly nice to use of broadcast syntax in places
  - Reimplements the C++ for the tiled case, though no pointers makes reasoning (and safety) better

Improvements!

- After CHEP we profiled the codes again
- We realised that there was significant time spent in the tiled algorithm in searching for the minimum d_{ij}
  - Although this needs to search over all jets, it is amenable to parallelism with a divide and conquer approach
    - Chunk the array in pieces, find the minimum d_{ij} in each part, then compare parts
- And that realising this optimisation was easy…
  - slight rewrite to use ternary operators
  - apply the `@turbo` macro from LoopVectorisation.jl

```
"""Find the lowest value in the array, returning th
find_lowest(dij, n) = begin
    best = 1
    @inbounds dij_min = dij[1]
    @turbo for here in 2:n
        newmin = dij[here] < dij_min
        best = newmin ? here : best
        dij_min = newmin ? dij[here] : dij_min
    end
    dij_min, best
end
```

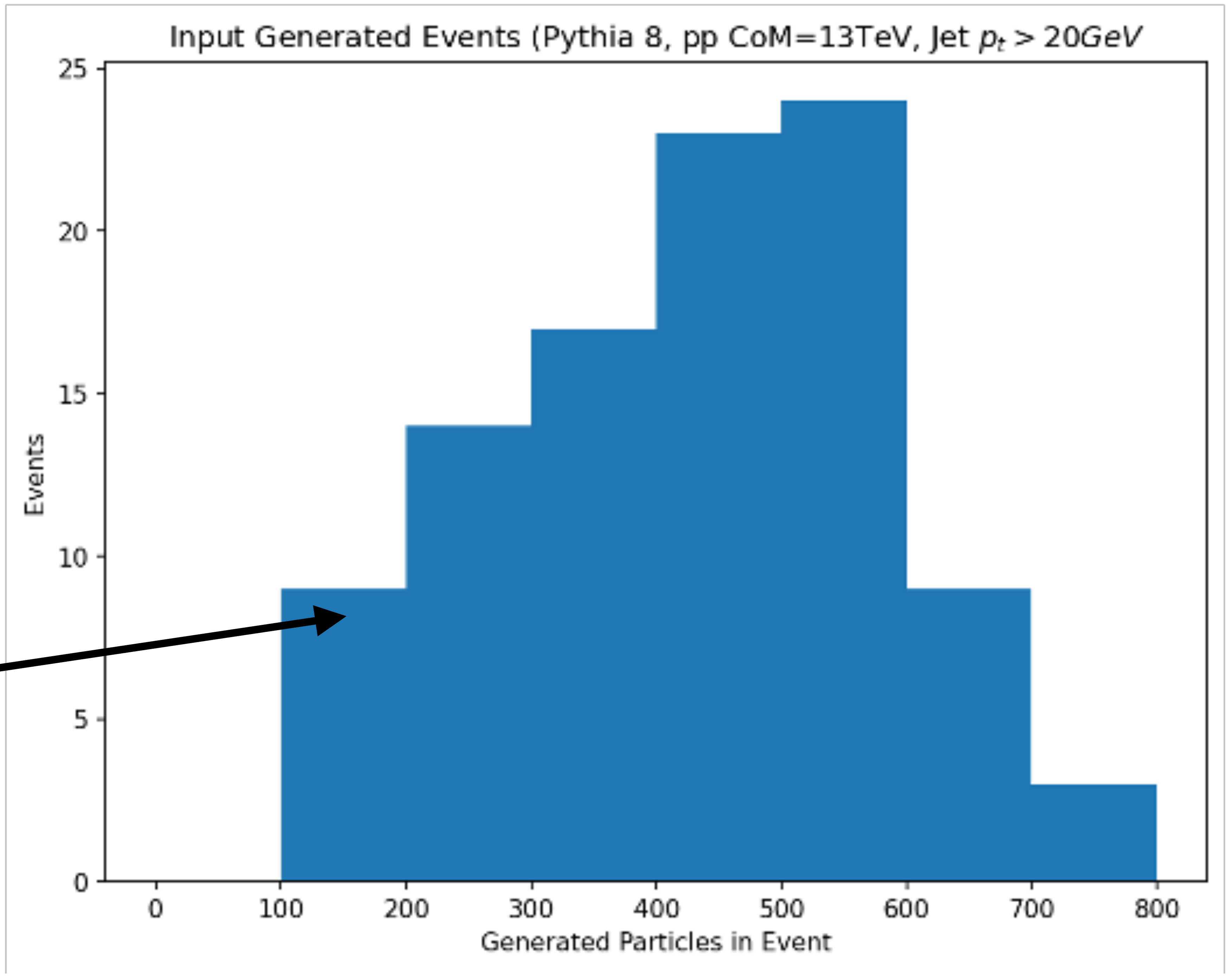
<u>This code is x5 faster</u> than `findmin()`!
28 μs vs. 131 μs
(See backup)

Other Algorithm Attempts

- Given that the use of SoA appeared to be so successful in the N2Plain case, wanted to try something similar in the tiled case
- Tried two different ways of doing this
  - Implement an SoA of jets for each tile
    - This turned out to be quite slow!
      - The main problem here was that allocating any collections for >500 tiles was just a killer for the overall time budget of ~200s μs/event
  - Have a global SoA structure for jets, with a simple linked list for the contents of each tile
    - This was faster than the per-tile SoA
    - But it was still slower than the original linked list N2Tiled
    - In the end, the tiled algorithm is so successful at reducing work that the parallelisation advantage of SoA was leached away
- Also, the coding of this was hard - definitely losing the ergonomic edge (although I didn't know about **StructArrays.jl** when I was doing this, that would have helped)

See strategies N2TiledSoAGlobal and N2TiledSoATile at JetReconstruction.jl@15bfd5

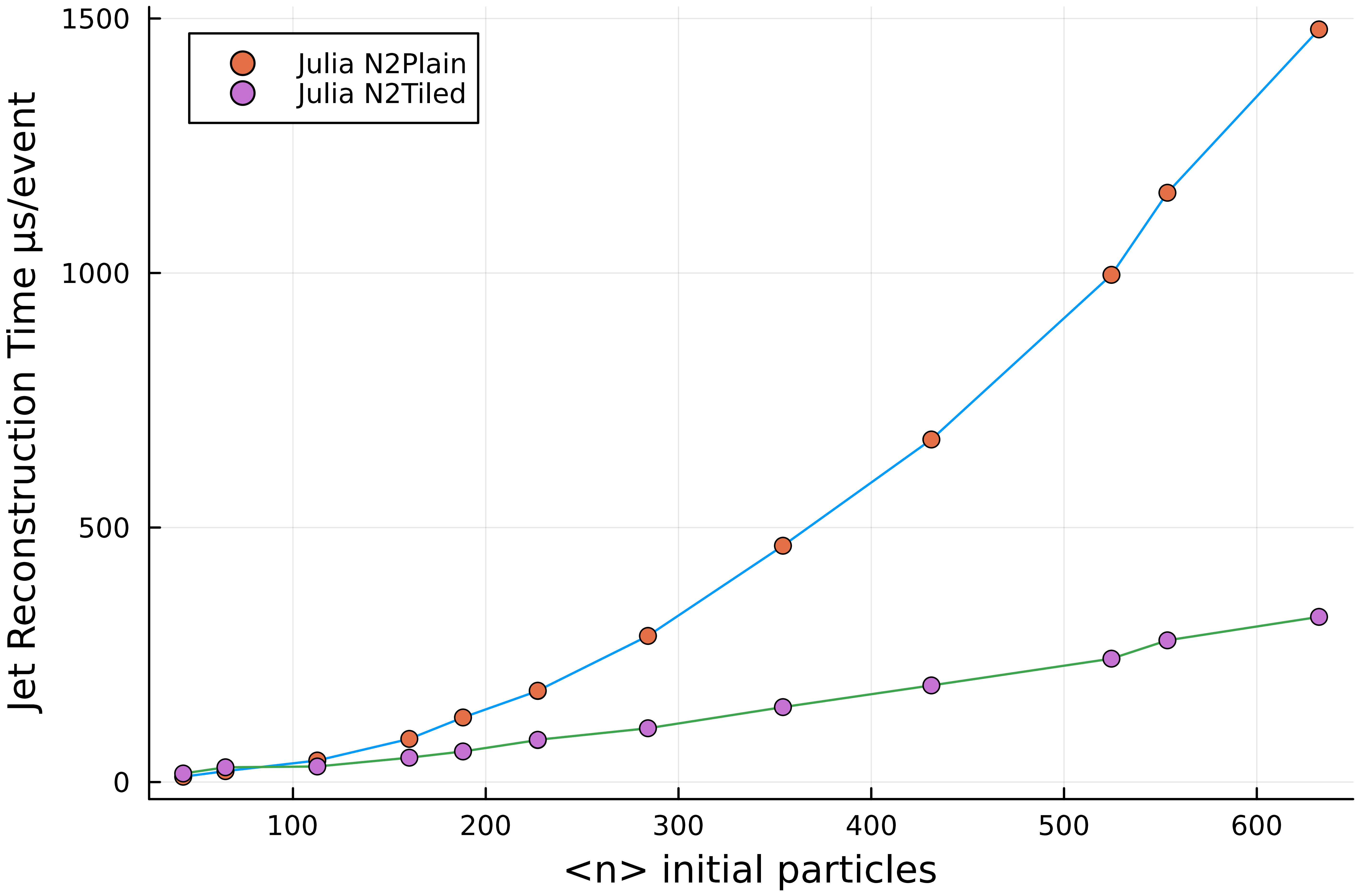
Current Performance



- Had been benchmarking the code with a sample of 100 13TeV pp events generated by Pythia8
  - Average initial particles 413
- Important to test performance at other working points
  - Generated additional samples over various ranges from <n> = 43 … 632
  - Plus a few heavy ion events (see backup)

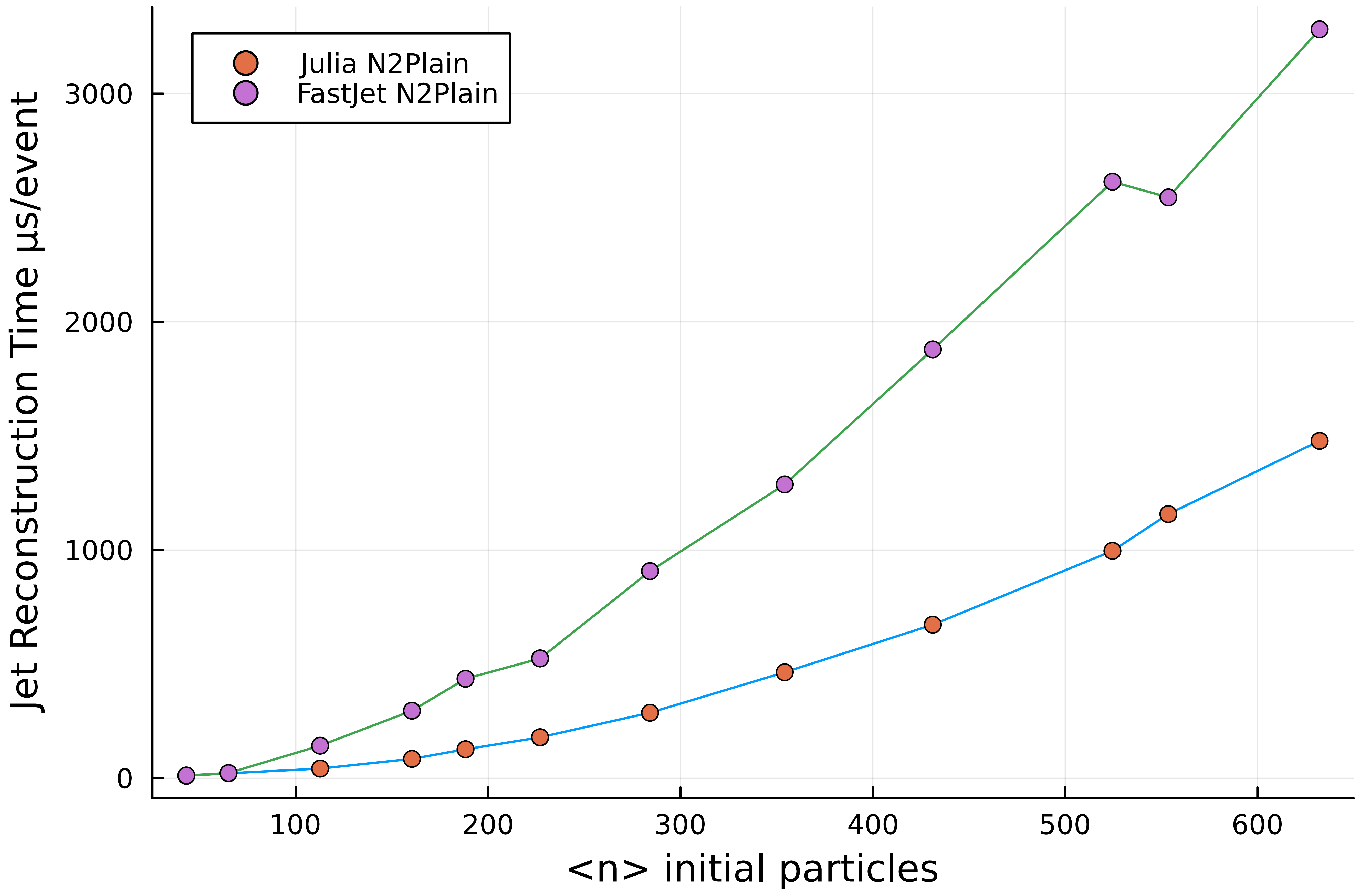
Julia Jet Finding

- Tiled algorithm strategy is very good and scales well
- Only at the lowest particle densities is the plain strategy better
  - e^+e^- Z: 37% faster
  - e^+e^- H: 25% faster



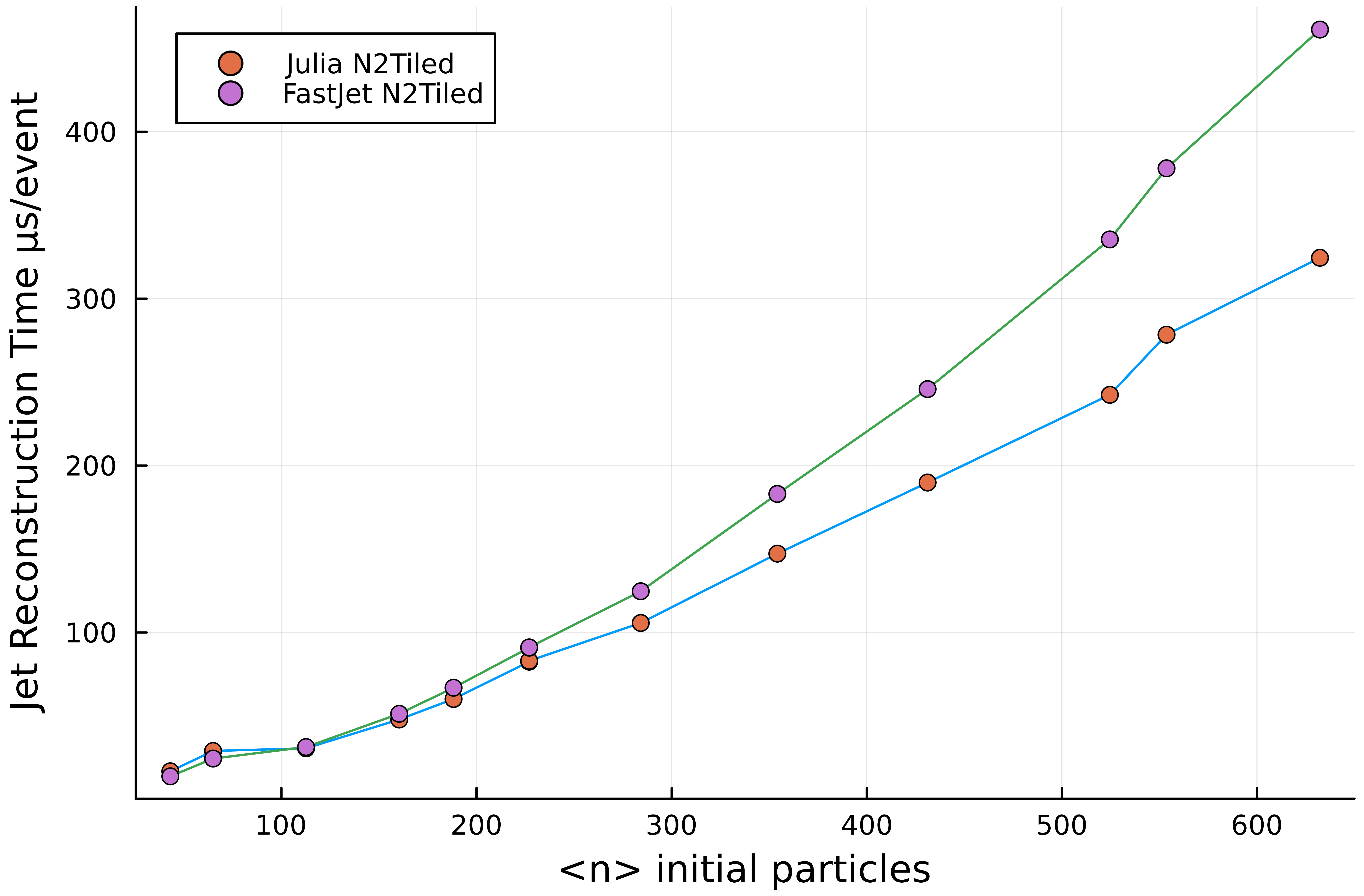
N2Plain: FastJet and Julia

- N2Plain scales a lot better in Julia
  - Structure of arrays and LoopVectorisation optimisation
- For e+e- Z pole events 13.5% faster
- To be fair to fastjet, one would not use this algorithm for N >~ 80
  - So not a regime to target for optimisation



N2Tiled: FastJet and Julia

- Small advantage for Julia at higher particle densities
  - This grows with density as the optimised `dij` finding is more significant
- However, the codes are pretty close
  - Main conclusion is that Julia reaches C++ speed
- Still would like to understand why without `@turbo` Julia is running a bit slower than FastJet



Preparing for Release - What is Done

- The Julia version here is fast enough to merit a release
  - Even if it's only a small fraction of what FastJet implements
- First make the interface for both implementations uniform:

```
function tiled_jet_reconstruct(particles::Vector{T};
    p = -1, R = 1.0, recombine = +, ptmin = 0.0) where {T}
```

- For the type T, we only require that the appropriate methods for E-p 4-vectors are defined
  - `pt2(), phi(), rapidity(), px(), py(), pz(), energy()`
  - Works fine with `LorentzVectorHEP` and `JetReconstruction.PseudoJet`
- Improved testing against FastJet as a reference (Anti-kT, Cambridge/Achen, Inclusive-kT)

Preparing for Release - Still TODO

- Write proper documentation
- Tidy up a few inconsistencies
  - Return consistent sequence merging history
  - Remove internal data member from PseudoJet
- Implement a “Best” strategy, dynamically switching based on <n>
- Fix plotting backend

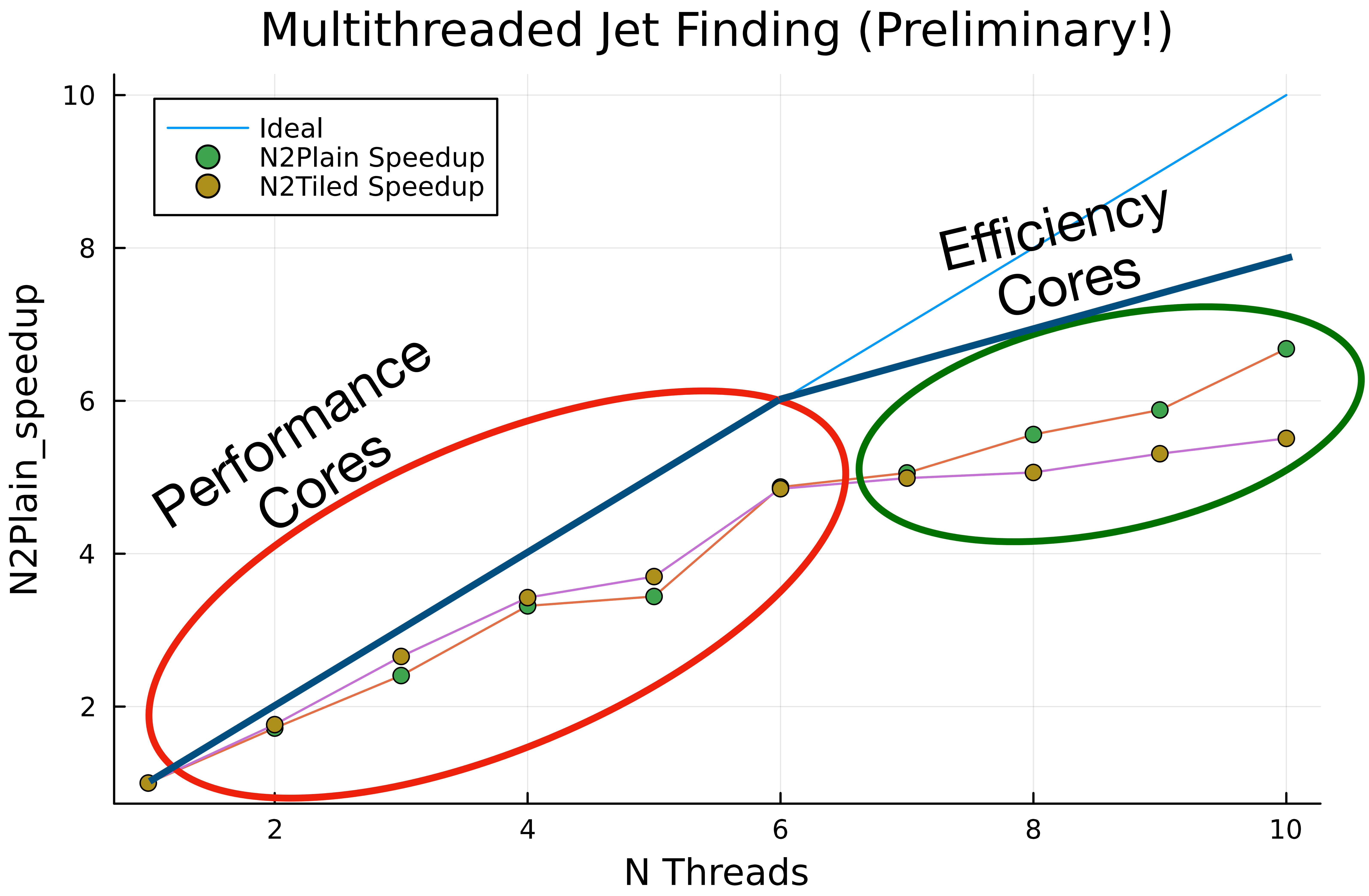
Conclusions

- Jet finding was an excellent example to try in Julia
- Performance was finally somewhat better than FastJet, which is known to be highly optimised
- Ergonomics of Julia were a lot better
  - No pointers: better memory safety and easier reasoning
  - Much easier to profile and to apply optimisations via macros
  - Tooling for debugging is a lot better
  - Much more flexible for users of the package to use their own datatypes
- Release of the package is rather close now

Should happen alongside a wrapped version of the FastJet C++

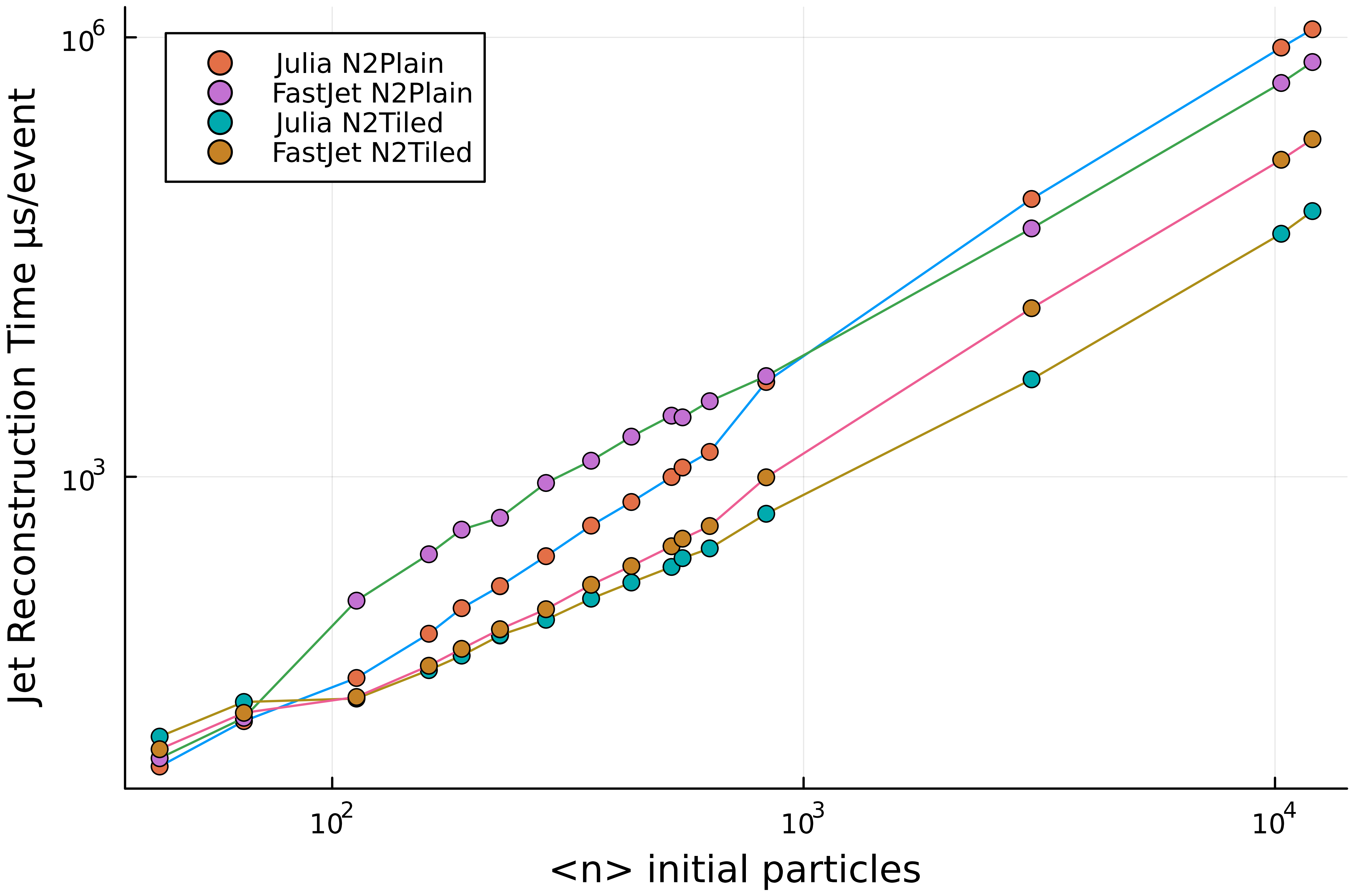
Backup

Multi-threading



Scaling is pretty good!

Very High Particle Densities (Heavy Ions)



- Suboptimal scaling of Julia N2Plain at very high densities to be understood
  - Not that it's actually a practical strategy at these particle densities

findmin() vs fast_findmin()

In [34]:

```
@benchmark for j in 450:-1:1 fast_findmin(x, j) end
```

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  26.250 μs …   1.311 ms  ┊ GC (min … max): 0.00% … 96.40%
 Time  (median):     26.875 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   28.079 μs ± 20.770 μs   ┊ GC (mean ± σ):  1.25% ±  1.67%
```


```
 Memory estimate: 14.06 KiB, allocs estimate: 450.
```

In [35]:

```
@benchmark for j in 450:-1:1 findmin(@view x[1:j]) end
```

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  125.542 μs …   1.908 ms  ┊ GC (min … max): 0.00% … 92.30%
 Time  (median):     127.250 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   131.795 μs ± 56.608 μs   ┊ GC (mean ± σ):  1.40% ±  3.04%
```


```
 Memory estimate: 49.22 KiB, allocs estimate: 1350.
```