Polyglot Jet Finding

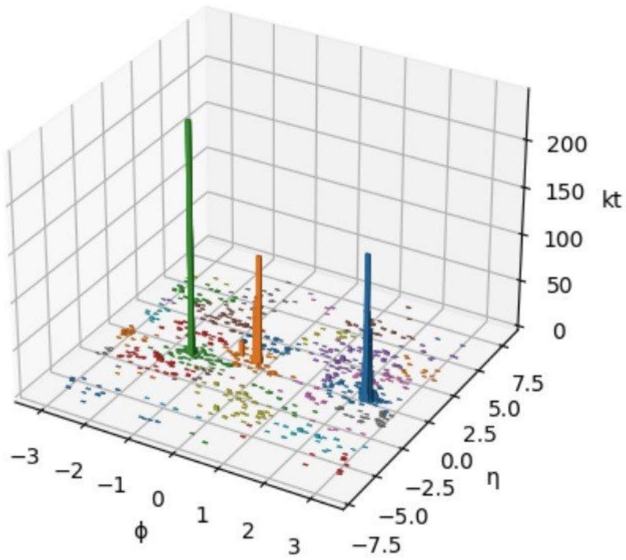
Graeme Stewart, Atell Krasnopolski, Philippe Gras, Benedikt Hegner

Overview

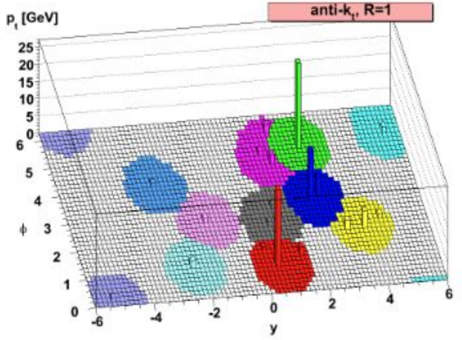
- Languages in HEP do evolve - albeit slowly!
  - Originally we programmed in Fortran for LEP
- With the LHC a huge transition to C++ occurred
  - Then supplemented by the addition of Python
    - Configuration and steering
    - Analysis codes
    - However, importantly backed by performant C++ code underneath
- However, there is interest over time in other languages (both inside HEP and outside)
  - Go attracted attention a few years ago
  - Julia is being actively instigated [link to other CHEP papers]
- Evaluation of any new language is multi-dimensional
  - Here we look at some aspects of algorithmic performance and lanugage ergonomics for current and possible future languages

AntiKt Jet Finding

- We would like to evaluate performance on 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
  - The AntiKt algorithm is popularly used because it is an infrared and co-linear safe algorithm
  - [arXiv:0802.1189]



AntiKt in Brief



1. Define a cone size R (0.4 is typical)
2. For each active pseudojet A (=particle, cluster)
   a. Measure the geometric distance, d, to the nearest active pseudojet B, if < R (or d=R)
   b. Define the AntiKt distance, akt_dist, as
      i. akt_dist = d / min(JetA p_t^2, JetB p_t^2)
      ii. N.B. Favours merges with high p_t jets, giving stability against soft radiation
3. Choose the jet with the lowest akt_dist
   a. If this jet has an active partner B, merge these jets
   b. If not, this is a final jet
4. Repeat steps 2-3 until no jets remain active

There is a parallelisation possibility in step 2

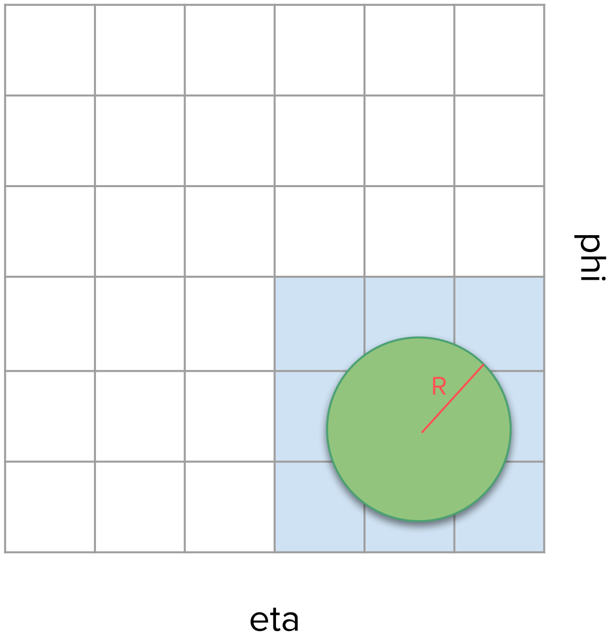
This is essentially a serial process (have to final the lowest global akt_dist)

Serial and Parallel Optimisations

- We look at two different approaches to this algorithm
  - A *basic implementation* of the algorithm, essentially just implementing the flow on the previous slide
  - A *tiled implementation* of the algorithm, where the (eta, 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

N.B. There is a FastJet C++ wrapper for both Python and Julia!

- The benchmark code used in HEP is FastJet in C++
  - This is a extremely well tested and optimised version
- Two versions in Python
  - One in pure Python
  - One using numpy and numba to accelerate calculations
- Julia version
  - Why Julia? Promise of the ergonomics of Python with speed approaching C++

Implementation	Basic Algorithm	Tiled Algorithm
C++ (FastJet)	TBD!	x
Python (Pure)	x	x
Python (Accelerated)	x	x
Julia	x	x

Ergonomics: C++

```
struct TiledJet {
  double     eta, phi, kt2, NN_dist;
  TiledJet * NN, *previous, * next;
  int        _jets_index, tile_index, diJ_posn;
};
```

- Tiles use pointers to jets
  - Implemented as a linked list
  - Minimises copying
  - Need to be careful about consistency with updating
    - Limited opportunities to parallelise
- Updates are bi-directional (jets are considered in pairs)
  - E.g., allows for a “march” across the tiles, only looking in one direction
- Overall, many pointers and linked lists make the code quite hard to follow

```
// Update of only RH neighbour tiles
for (Tile ** RTile = tile.RH_tiles; RTile != tile.end_tiles; RTile++) {
  for (jetA = tile.head; jetA != NULL; jetA = jetA->next) {
    for (jetB = (*RTile)->head; jetB != NULL; jetB = jetB->next) {
      double dist = _tj_dist(jetA,jetB);
      if (dist < jetA->NN_dist) {jetA->NN_dist = dist; jetA->NN = jetB;}
      if (dist < jetB->NN_dist) {jetB->NN_dist = dist; jetB->NN = jetA;}
    }
  }
}
```

Ergonomics: Pure Python

```python
def scan_for_all_nearest_neighbours(jets: list[PseudoJet]):
    '''Do a full scan for nearest (geometrical) neighbours'''
    for ijetA, jetA in enumerate(jets):
        for ijetB, jetB in enumerate(jets[ijetA+1:], start=ijetA+1):
            dist = geometric_distance(jetA, jetB)
            if dist < jetA.info.nn_dist:
                jetA.info.nn_dist = dist
                jetA.info.nn = ijetB
            if dist < jetB.info.nn_dist:
                jetB.info.nn_dist = dist
                jetB.info.nn = ijetA
        jetA.info.akt_dist = antikt_distance(jetA, jets[jetA.info.nn] if jetA.info.nn else None)
```

- Easy implementation of jet classes
- Using a simple list to hold pseudojets
  - Mutable, so updates are easy
- Logic is clear and overall the implementation takes up relatively few lines of code in the basic algorithm case
- Tililed algorithm makes things more complicated, but still a fairly straightforward implementation, with simpler data structures used

Ergonomics: Accelerated Python

```python
class NPPseudoJets:
    def __init__(self, size:int):
        '''Setup blank arrays that will be filled later'''
        self.size = size
        self.phi = np.zeros(size, dtype=float)          # phi
        self.rap = np.zeros(size, dtype=float)          # rapidity
        self.inv_pt2 = np.zeros(size, dtype=float)      # 1/pt^2
        self.dist = np.zeros(size, dtype=float)         # nearest neighbour geometric distance
```

- Using numba to hold arrays for pseudojets
  - Basically a single structure of arrays object
- Calculations can be aggressively parallelised for basic case
- Bookkeeping has to be done with masks to avoid resizing
- Numba jitting needs basic numpy types (unless taught otherwise)
- For the tiled case, used a single unified array in [i_{eta}, i_{phi}, SLOTS]
- Needs to be sized appropriately Leading to wasted space
- Ironically, parallelisation suffers in this algorithm version

```python
@njit
def scan_for_all_nearest_neighbours(phi: npt.ArrayLike, rap: npt.ArrayLike, inv_pt2: npt.ArrayLike,
                                    dist: npt.ArrayLike, akt_dist: npt.ArrayLike,
                                    nn: npt.ArrayLike, mask:npt.ArrayLike, R2:float):
    '''Do a full scan for nearest (geometrical) neighbours'''
    for ijet in range(phi.size):
        if mask[ijet]:
            continue
        _dphi = np.pi - np.abs(np.pi - np.abs(phi - phi[ijet]))
        _drap = rap - rap[ijet]
        _dist = _dphi*_dphi + _drap*_drap
        _dist[ijet] = R2 # Avoid measuring the distance 0 to myself!
        _dist[mask] = 1e20 # Don't consider any masked jets
        iclosejet = _dist.argmin()
        dist[ijet] = _dist[iclosejet]
```

arrays!

Ergonomics: Julia

array

```
_kt2 = 1.0 / (JetReconstruction.pt.(_objects) .^ 2)
```

- Uses broadcast syntax for array calculations
- Easy markup where SIMD can be used as well
- Keeps the code for the basic implementation rather nice, easy to follow
- For the tiled case, the implementation follows fastjet
  - Using references, not pointers
- Jitting takes a few seconds (on my machine) for the tiled case
  - Borderline annoying when making rapid iterations cf. Python (but less than C++ compilation!)

```
@inbounds @simd for j in from:(i-1)
    Δ2 = _dist(i, j, _eta, _phi)
    f = Δ2 <= nndist
    nn = IfElse.ifelse(f, j, nn)
    nndist = IfElse.ifelse(f, Δ2, nndist)
end
```

Runtime Speed

- Standard sample 100 of HepMC3 events, multiple trials
- Benchmark is C++ Tiled Algorithm at 299µs/event (=1.00)
- Jit time for Numba and Julia is excluded

Implementation	Basic Algorithm	Tiled Algorithm
C++ (FastJet)	TO BE DONE	1.00
Python (Pure)	779	177
Python (Accelerated)	31	175
Julia	2.8	1.1

Python acceleration of tiled algorithm currently doesn't give a speed-up
[I think it can be improved]

Bonus Observations

- Pure Python 3.11 is much faster than 3.10
  - Pure python basic and tiled run 30% faster in 3.11
- Squeezing maximum performance from Julia requires some tricks, e.g.,
  - Switching off array bounds checking
  - Paying attention to memory allocations, e.g., in loops
  - Profiling some occasional fumbles from the jit (e.g., pow(x,-1.0) instead of 1/x)
  - (However, performance is usually excellent out of the box)

Conclusions

- FastJet in C++ remains the champion of speed!
  - However, the code is tricky and not easy to work with
- The pure Python implementation has the advantages of working in a easy language
  - However, its runtime speed is, as expected very poor
- The accelerated Python implementation sacrifices ergonomic advantages, moving to array structures
  - The speed-up in the basic case is significant
  - The speed-up in the tiled case is disappointing
    - Numpy excels at parallel calculations, but the tiling implementation is not optimal for this
- Julia looks impressive, it’s easy to work with and fast
  - “Time to first plot” is quite high because of the JIT compilation
  - Features like array broadcast really help for the basic implementation

Backup

Repositories

Implementation	Repository
C++	https://fastjet.fr/
Python (all)	https://github.com/graeme-a-stewart/antikt-python
Julia Basic	https://github.com/JuliaHEP/JetReconstruction.jl
Julia Tiled N^2	https://github.com/grasph/AntiKt.jl

Benchmark Machine

- 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
- Ubuntu 22.04 running in WSL under Windows 11