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 wholesale transition to C++ occurred
  - Then supplemented by the addition of Python in specific areas
    - 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)
  - Java had its aficionados, even as C++ was on the rise
  - Go attracted attention a few years ago
  - Julia is being actively investigated [CHEP2023: Tamás talk, Jerry talk]
- Evaluation of any new language is multi-dimensional
  - Here we look at some aspects of algorithmic performance and language ergonomics for different 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]
FastJet AntiKt in Brief

1. Define a distance parameter $R$ (0.4 is typical)
   a. This is a “cone size”

2. For each active pseudojet A (=particle, cluster)
   a. Measure the geometric distance, $d$, to the nearest active pseudojet B, if $< R$ (else $d=R$)
   b. Define the AntiKt distance, $akt\_dist$, as
      i. $akt\_dist = d \times \min(JetA p_t^2, JetB p_t^2)$
      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

Step 3 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

- 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++ ([see previous talk!](#))

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Basic Algorithm</th>
<th>Tiled Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ (FastJet)</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Python (Pure)</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Python (Accelerated)</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Julia</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

N.B. There is a FastJet C++ wrapper for both **Python** and **Julia**
Ergonomics: C++

- FastJet code is very C-ish, for speed
  - Pretty well written code
- Tiles use pointers to jets
  - Implemented as a linked list
  - Minimises copying
  - Need to be careful about consistency with updating
    - Limited opportunities to parallelise
- Overall, many pointers and linked lists make the (tiling) code quite hard to follow

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

```c
// 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

- 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

```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)
```
Ergonomics: Accelerated Python

- 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 \([i_\text{eta}, i_\text{phi}, \text{jet}]\)
- Needs to be sized appropriately (many empty slots)
- Parallelisation suffers a lot in this algorithm version
Ergonomics: Julia

- Uses broadcast syntax for array calculations
- Easy markup for extra SIMD hints can be used as well
  - Nice built in profiler!
- 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. pure Python (similar to numba jit, but less than C++ compilation!)

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

@inbounds @simd for j in from:(i-1)
  Δ2 = _dist(i, j, _eta, _phi)
  if Δ2 < nndist
    nn = j
    nndist = Δ2
  end
end
```
Runtime Speed

- Standard sample **100 of Pythia8** events pp 13TeV, jet $p_t > 20\text{GeV}$, multiple trials
- Benchmark is C++ Tiled $N^2$ Algorithm 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

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Basic Algorithm</th>
<th>Tiled Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ (FastJet)</td>
<td>17.6</td>
<td><strong>1.00</strong></td>
</tr>
<tr>
<td>Python (Pure)</td>
<td>966</td>
<td>222</td>
</tr>
<tr>
<td>Python (Accelerated)</td>
<td>53.4</td>
<td>178</td>
</tr>
<tr>
<td>Julia</td>
<td>4.00 😃</td>
<td><strong>1.12</strong></td>
</tr>
</tbody>
</table>

Python “acceleration” killed by parallelism reductions for tiled algorithm

Julia finds an exploits SIMD optimisations in loops

Good speed up

Julia 12% off FastJet tiled
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 does require some tricks, e.g.,
  - Paying attention to memory allocations, e.g., in loops
  - Profiling - we did see some occasional fumbles from the jit
    - e.g., pow(x,-1.0) instead of 1.0/x, though this doesn’t happen in current versions
  - Switching off array bounds checking, giving simd hints
    - @inbounds @simd gains ~35%
    - However, even without these hints Julia is x2.5 faster than C++ for the basic algorithm - it finds many optimisations without hints
    - i.e., performance is excellent ‘out of the box’
Conclusions

- FastJet in C++ remains the champion of speed!
  - However, the code is tricky and not so 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 pretty terrible (at least for what we tried)
    - Numpy excels at parallel calculations, but the tiling implementation is not optimal for this
- Julia is impressive, it’s easy to work with and fast
  - “Time to first plot” is an issue because of the JIT compilation
    - No worse than numba and much improved in the next release (1.9)
  - Features like array broadcast and loop vectorisation really help
Backup
## Repositories

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td><a href="https://fastjet.fr/">https://fastjet.fr/</a></td>
</tr>
<tr>
<td>Python (all)</td>
<td><a href="https://github.com/graeme-a-stewart/antikt-python">https://github.com/graeme-a-stewart/antikt-python</a></td>
</tr>
<tr>
<td>Julia Tiled N²</td>
<td><a href="https://github.com/grasph/AntiKt.jl">https://github.com/grasph/AntiKt.jl</a></td>
</tr>
</tbody>
</table>
Benchmark Machine

- Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
- CentOS Stream 8 OS

- Fastjet compiled with gcc 8.5.0, -O2
- Python 3.10.10, numpy 1.23.5, numba 0.56.4
  - Python 3.11.0 also tested for pure Python codes
- Julia 1.8.5
Input Event Sample

- **Generated** with Pythia 8, pp collisions at 13TeV
  - Cut applied for minimum jet $p_t$ of 20GeV