21 Dec 2022

Lukas Arnold
Brendan Cunnie

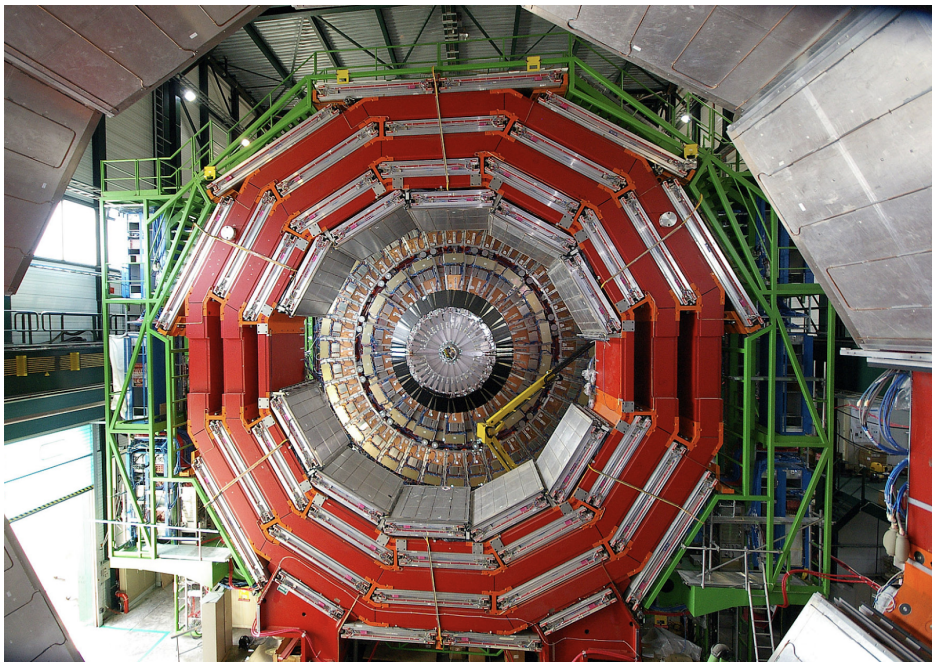
Particle Tracker Report

Project Description

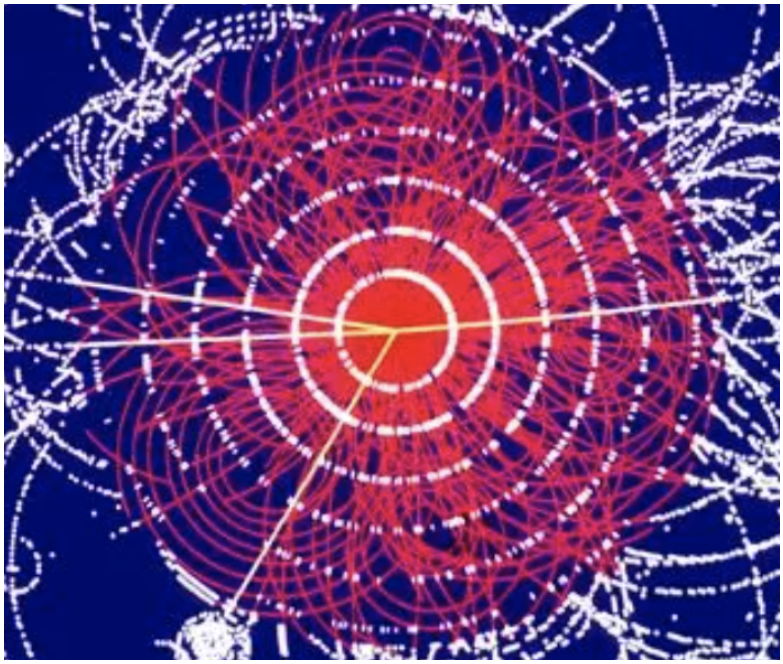
Inspired by particle detectors at the Large Hadron Collider, our project is dedicated to a computational problem in particle physics: track finding.

Particle collisions create an immense amount of collision products – particles – that are captured by the detectors surrounding the collision point. The particle tracks are identified as pixels by the detectors; in order to identify the particle, the possible tracks have to be reconstructed. Due to the immense amount of particles, parallelization of track finding seems to be an appropriate answer to the computational load.

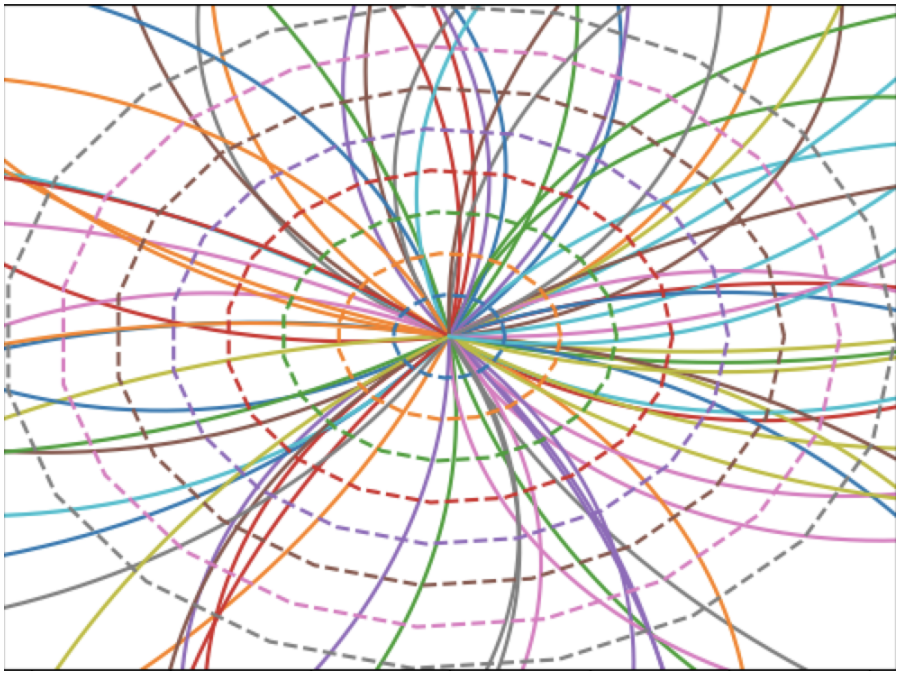
In the picture below, the CMS particle detector at the Large Hadron Collider at CERN, Geneva, is shown. Different, layered segments of the detector are visible.



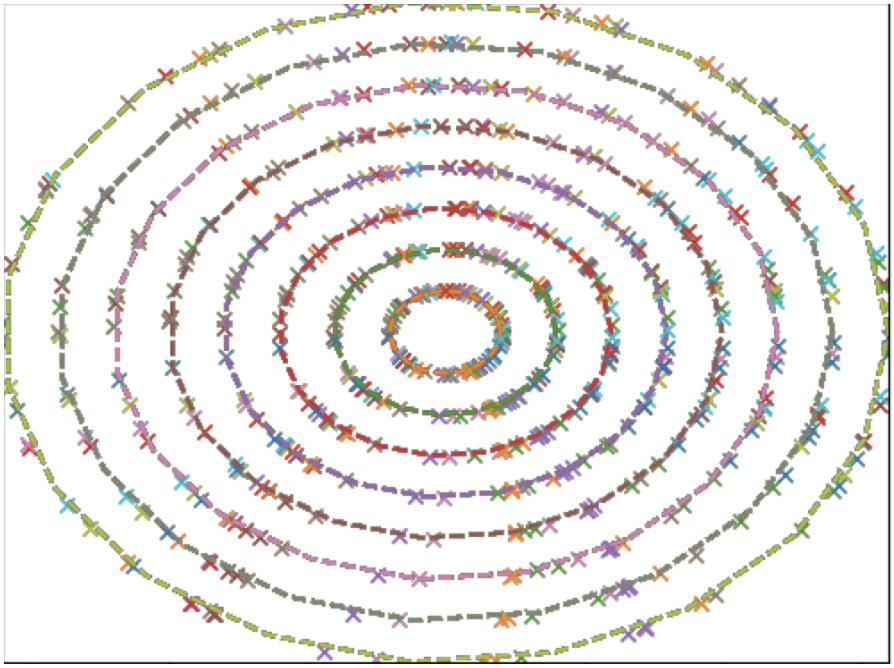
The next picture shows an actual reconstructed collision at the CMS detector, showing particle tracks in red and deposited energies (omitted in our emulation) in White.



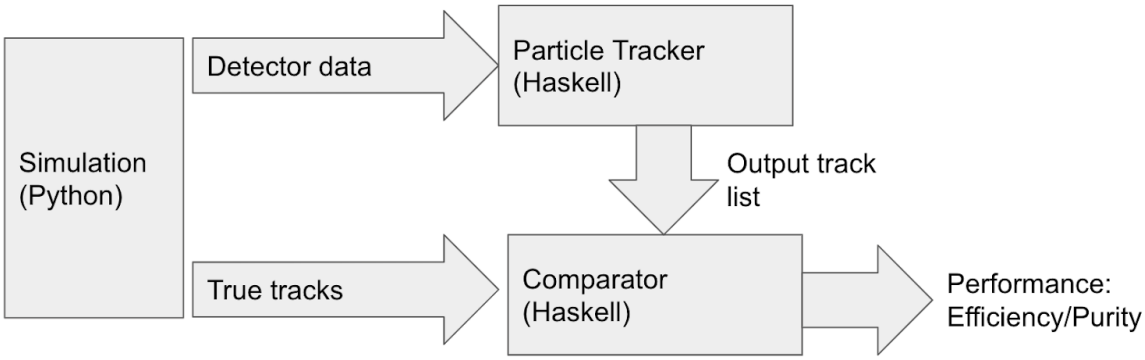
The next picture shows our emulation of a simplified version of the particle detectors. It consists of multiple layers of pixels that detect particles passing through. The layers have all same angular resolution, i.e. distinct spatial resolution. Each particle is bent by a magnetic field whereas the radius is determined by the energy of the particle.



The detector “sees” what is shown on the picture below:



Dataflow



The dataflow consists of multiple parts:
The simulation outputs both the “true tracks” as well as the data that the detector sees; the detector data is input to the particle tracker that creates a list of potential particle tracks. These are compared against the true tracks by a comparator that evaluates the performance in terms of efficiency and purity, which in our case was brought into an acceptable range (efficience above 90%, purity above 3%).

Algorithm

1. Take points on first layer
2. For each point on first layer, calculate distances between that point an any point on second layer
3. Extrapolate, and include any point that might fit the track
4. After we create the list of all speculative tracks, we run through a filter to keep only the tracks whose path approximates an arc.

Parallelization

For parallelization we made three general changes

- Judiciously used parMap for map,
- and parList on List Comprehensions.
- Created new function parFilter.

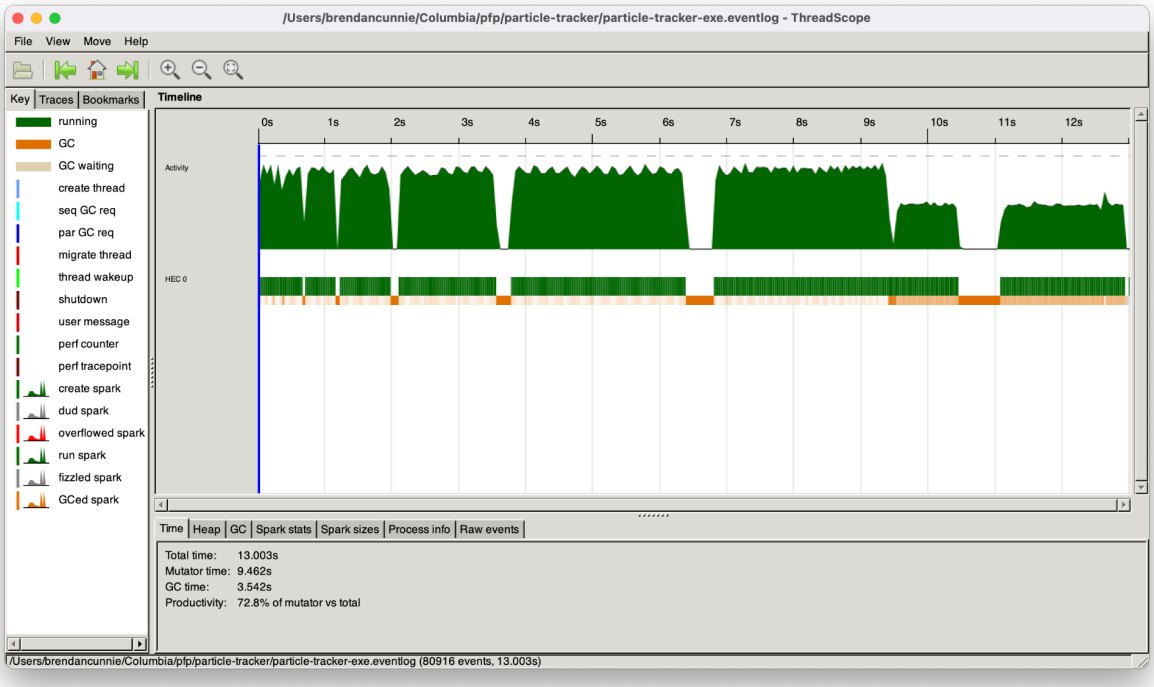
We discovered we could not use parMap or parList indiscriminately. For most list comprehensions, using parList actually made the code *slower*, even with 8 CPUs.
So it was not uncommon to make a change to parallelize, test the results, and then revert it.

In addition we made a new function parFilter to replace filter with a parrallelized function with a similar signature. This was used to take the (very long) list of proposed tracks, and filter them down to just the tracks that approximate arcs or linear paths.

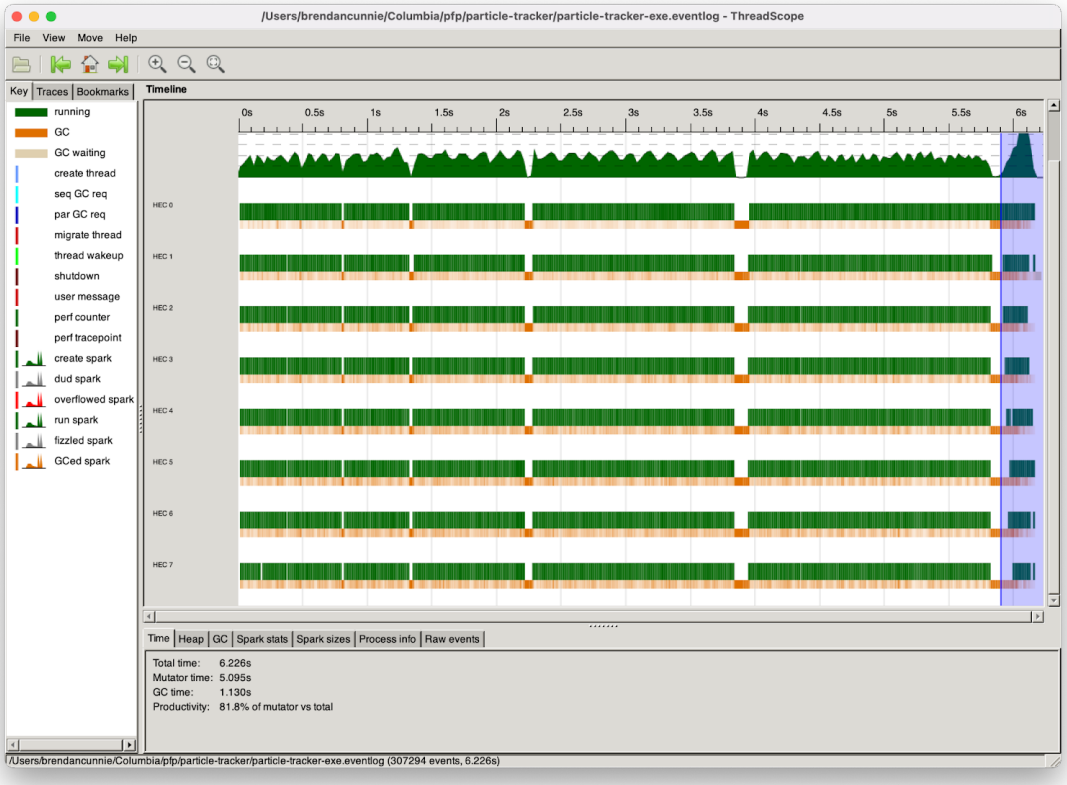
parFilter works by breaking the list into chunks, distributing the chunks to the CPUs, and concatenating the filtered results. We experimented with chunks of various sizes. The fastest (or near fastest) turned out to be to break the list into as many chunks as we have CPUs (`numCapabilities`). We expected smaller chunks to be more efficient à la Sudoku. But it looks like the filtering logic did not have much variation in CPU time, so we never saw any CPU overloaded or underloaded. At the other extreme, trying very small chunk sizes resulted in very poor performance.

parFilter's helper function does *not* combine partial lists as it filtered through the tracks. It return a list of lists so parFilter can concatenate all at once.

Overall we saw 11-13 seconds processing on a single CPU.



But we saw a little over 6 seconds over eight CPUS, roughly doubling the performance.



That said, parFilter by itself saw a roughly 3 times improvement in speed (0.5 seconds vs 1.5 seconds.) The initial list generation uses hierarchical processing, which does not distribute as well. Whereas with filtering, each filter is wholly independent.

.gz Manifest

testcase_generator.py -- generates inputdata.csv, the raw detector data

app/Main.hs -- generates the long list of speculative tracks
src/parFilter.hs -- a parallelized version of the filter function
src/Lib.hs -- numerous helper functions used to validate each speculative track

app/test.hs -- comparator and performance measurement

inputdata.csv -- a boolean 2D array indicating where detectors detected particles
output.csv -- output file listing tracks that approximate arc or linear tracks