Optimization of Annealing Pattern Recognition Algorithms applied to Charged Particle Track Reconstruction

Student - S.Conlon, **Mentor** - P.Calafiura

August 8, 2019

Abstract

The purpose of this research was to investigate methods for optimizing the run time efficiency of the seeding step for quantum or simulated annealing pattern recognition algorithms for track reconstruction on the ATLAS experiment at CERN. Three seeding algorithms were tested using a standard laptop hardware configuration and on a supercomputing cluster. The fastest algorithm tested utilizes parallel computing techniques as well as runtime compilation to run 4.68-times faster than the original algorithm on the laptop and 24.56-times faster on the supercomputer. The speed-up demonstrated by this study will have an impact on future annealing pattern recognition projects as well as machine learning research applied to charged particle track reconstruction.

1 Introduction

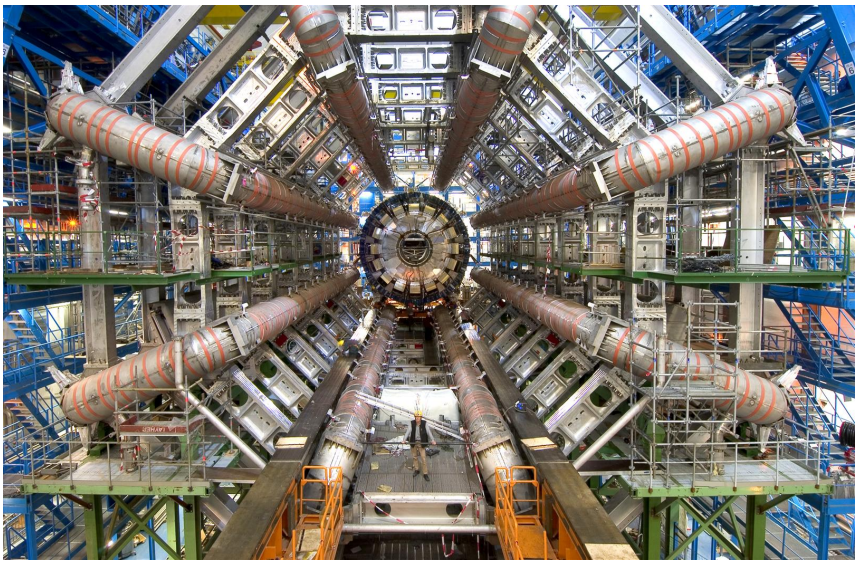


Figure 1: The ATLAS experiment at CERN under construction

The ATLAS experiment at CERN measures the products of 13 TeV proton-proton collisions at the Large Hadron Collider (LHC). Every 25ns the ATLAS detector readout systems captures a snapshot ("Event") of the status of the detector. Each Event contains the measurements of ≈ 40 p-p collisions, and each collision produces ≈ 30 long-lived charged particles. These particles then move outwardly through the detector apparatus and each layer of the detector records the position of the particle at that instant of time, along with other physical properties. The mission of track reconstruction is to take the data collected from the particle detectors and connect these dots together in order to reconstruct the paths the particles took when they exited the detector. These tracks can then later be examined for interesting physics.

Around 2026 the High Luminosity LHC (HL-LHC) upgrade will come online enabling, among other things, precision physics studies in the Higgs sector. The high luminosity improvements to the LHC will generate 5-10 times more high energy collisions than seen previously, and upgrades to the ATLAS Trigger and Data Acquisition (TDAQ) system will allow to process 5-10 times more events per second. While this means more interesting physics, it also puts a heavier load on ATLAS's existing track reconstruction software. The traditional algorithms for track reconstruction are estimated to scale at greater than $O(N^2)$ run time as the number of collisions, N, increases. For this reason, along with technological and funding constraints, experts estimate that there will be a shortage of computational resources by a factor of three to ten times [1]. So it is important to search for new, faster algorithms to solve the track reconstruction problem.

The difficulty of the problem encourages us to explore unconventional approaches. It has been shown that quantum computers can more efficiently identify patterns in large sets of data. [2] In order to take advantage of this gain in efficiency, the track reconstruction problem must be reformulated in terms of a pattern recognition problem. This can be done by mapping the set of possible tracks into a Quadratic Unconstrained Binary Optimization problem. Finding the minimum of this function can be done on a quantum computer or a classical computer, using a simulated quantum environment. This approach was tested by the ATLAS group at Lawrence Berkeley National Laboratory using a D-Wave computer and found that the quantum computer was able to solve the track reconstruction problem with comparable precision to state-of-the art tracking algorithms. [3]. However, the run time of the D-Wave quantum computer was limited by

[1] Antonio Augusto Alves Jr, "A Roadmap for HEP Software and Computing R and D for the 2020s," Cornell University arXiv. 1712.06982

[2] Ralf Schültzhold, "Pattern recognition on a quantum computer," Physical Review. 67, 062311-1 - 062311-6

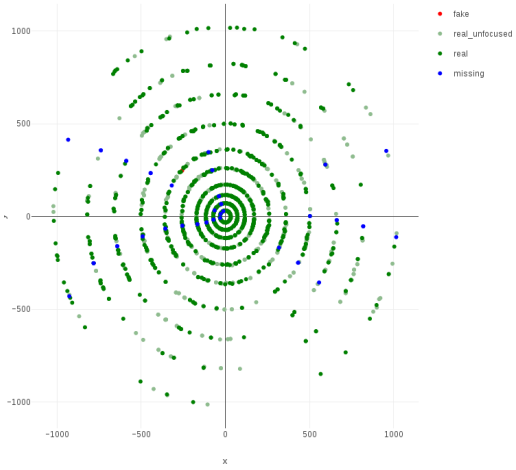
[3] Lucy Linder, "Using a Quantum Annealer for particle tracking at LHC,"(Master's thesis). University of Applied Sciences and Arts Western Switzerland

the time needed to process the data before it could be input onto the machine. This action of preparing the data for the quantum annealer is called seeding. The goal of this research was to optimize the runtime performance of the seeding algorithm.

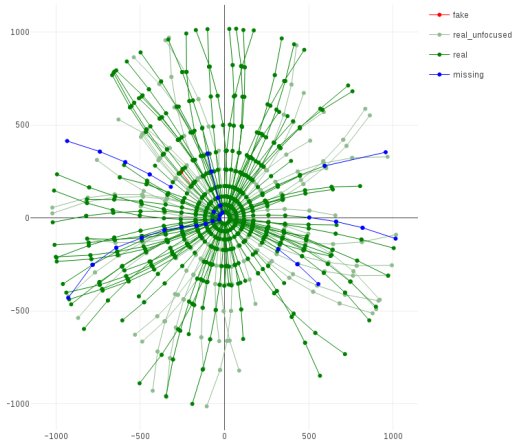
2 Experimental Setup and Methods

2.1 Code Base

Qallse is the quantum pattern recognition software which was used as the basis for the optimization study. The code base is open source and was developed by Lucy Linder and the ATLAS computation group at LBNL.[4]



(a) Input hits into Qallse



(b) Output tracks from Qallse

2.2 Detector Set Up

For the purposes of the study, an open-access dataset of MC simulated Events based on the design of a generic HL-LHC particle tracking detector was used[5]. The detector geometry consists of ten concentric cylinders which constitute the pixel detector layers. Each hit belongs to one of these ten layers. The standard coordinate system for ATLAS is shown in Figure 3. The interaction zone encapsulates the area in which proton collisions occur. This region is important to track reconstruction because we are only interested in particles which originate from this zone. The interaction zone is defined as a 700 mm long cylinder along the z-axis, centered on the origin. The cartesian coordinates are as follows: the z-axis points along the beam line, the positive x-axis points toward the center of the LHC accelerator ring, and the positive y-axis points upward. The cylindrical coordinates are as follows: the radial dimension, R, is the distance from the center of the interaction region, the azimuthal angle, phi, is the measured in the x-y plane starting from the positive x-axis, the polar angle, theta, is measured in the x-z plane starting from the positive z-axis.

[4] https://github.com/derlin/hepqpr-qallse
[5] https://sites.google.com/site/trackmlparticle

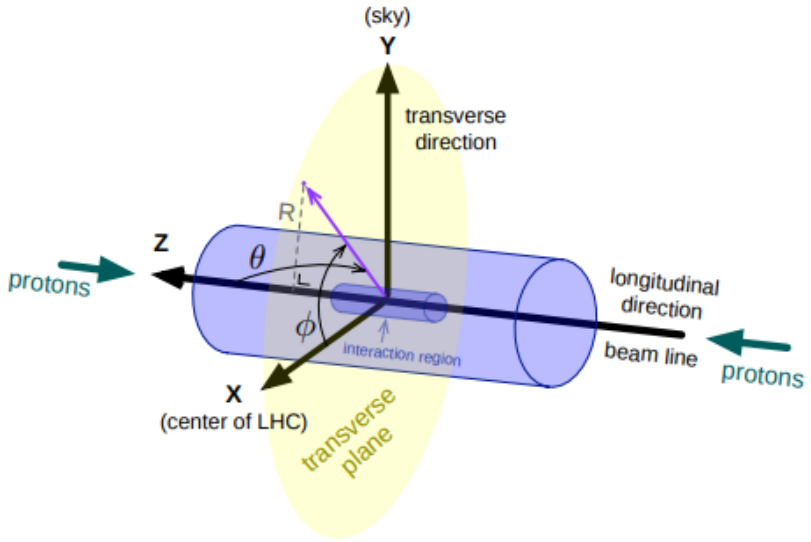


Figure 3: The ATLAS coordinate system

2.3 Overview of Seeding Algorithm Design

The goal of seeding is to take the set of particle hits in the detector and construct a list of all possible doublets. A doublet is a pair of consecutive hits belonging to the same particle. The algorithm can be broken down into four main parts:

1. Select an inner hit, the first hit in the doublet, from the set of all hits in the detector.
2. Using the position of this inner hit, calculate a region of interest in which the outer hit could exist, the second hit in the doublet.
3. Move through the rest of hits in the detector and check if they fall within our calculated region. If a hit falls within this region, then check if the inner/outer hit pair satisfies a list of physical requirements. Then, if the inner/outer hit pair satisfies these requirements then it should be added to the list of possible doublets.
4. Repeat from step one until all hits in the detector have been used as inner hits.

At the end of this process, the algorithm has successfully generated a list of doublets that satisfy our requirements.

2.3.1 Calculating the Region of Interest

Given the inner hit to a doublet, the seeding algorithm must calculate a three-dimensional "Region of Interest" within the detector where possible outer hit candidates could be contained. First, the outer hits must be contained in layers that are between 10 mm and 300 mm away from the inner hit. Second, the phi direction is sliced into 53 discrete slices and outer hits must be contained in the same or adjacent phi slices as the inner hit. Third, the line which is formed by the inner and outer hit pairs must intersect with the region of interest. This requirement is used to limit the range in which outer hits can be placed in the z-direction. A diagram of the region of interest is shown in Figures 4 and 5. The equation for calculating the z range is

$$minInterest = zMinus + R_{layer} * \frac{Z_{inner} - zMinus}{R_{inner}} \tag{1}$$

$$maxInterest = zPlus + R_{layer} * \frac{Z_{inner} - zPlus}{R_{inner}} \tag{2}$$

Once the region of interest for possible outer hit candidates has been calculated, the algorithm can then use these ranges to more efficiently filter through the outer hits.

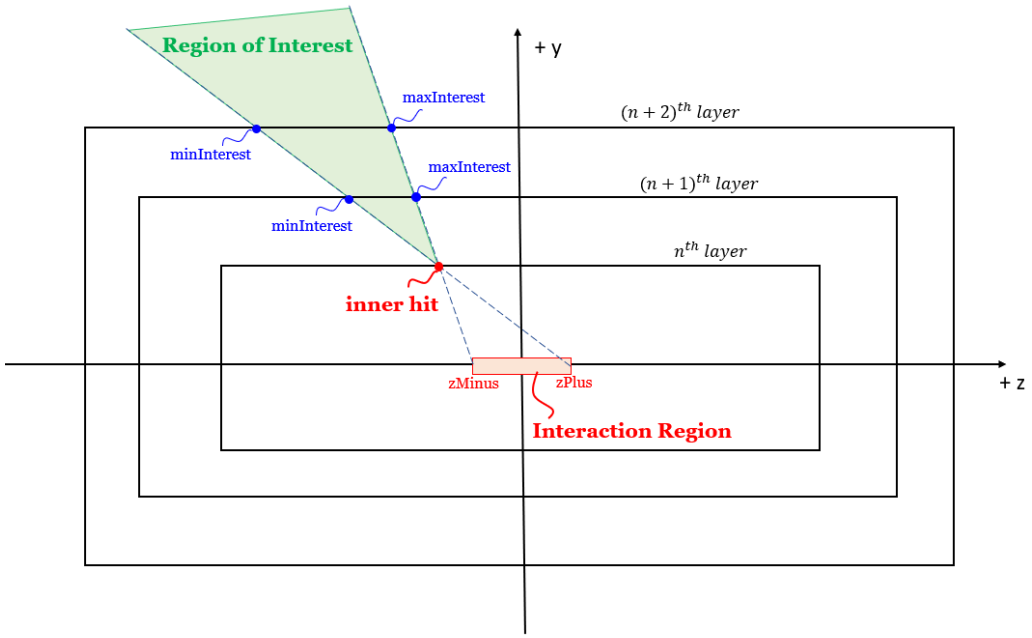


Figure 4: Diagram showing the region of interest bound in the z-direction for a given inner hit. Note, each layer has its own z range.

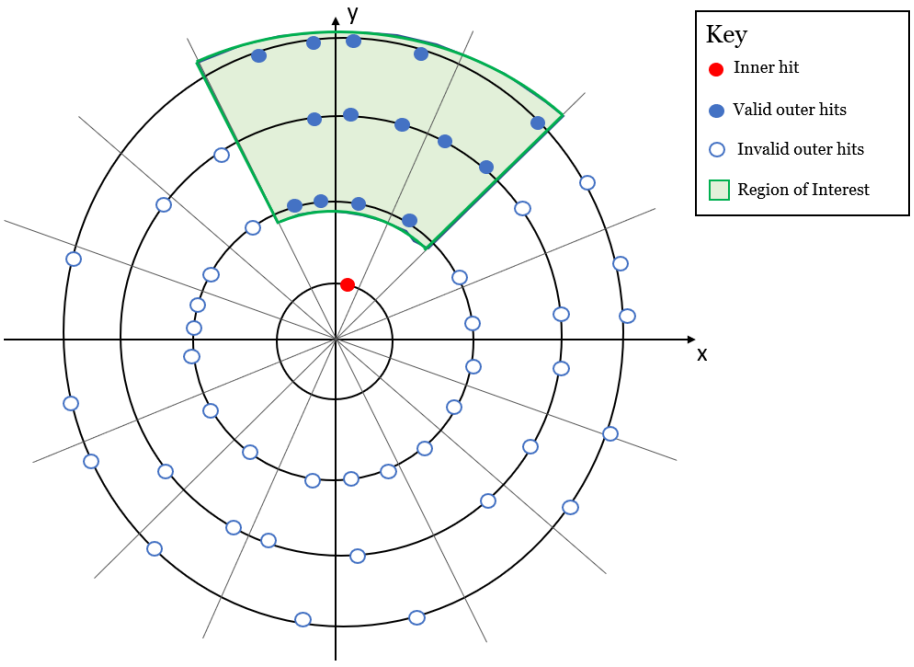


Figure 5: Diagram showing the region of interest projection on the x-y plane for a given inner hit. Note, only outer hits inside the region of interest are valid.

2.3.2 Filtering Outer Hits

The algorithm now iterates through the set of possible outer hits and applies a series of filters in order to determine if the inner/outer hit pairs form physically viable doublets. If an outer hit

passes all the filters, then the inner/outer hit pair is added to the list of possible doublets The following filters are applied as follows:

- Filter Layers: this function returns true if the outer hit belongs to a layer in the region of interest.
- Filter Phi: this function returns true if the outer hit belongs to a phi slice in the region of interest.
- Filter Z: this function returns true if the outer hit is in the z range for the region of interest.
- Filter Doublet Length: this function returns true if the distance between the inner and outer hits is between 10 mm and 300 mm.
- Filter Horizontal Doublets: this function returns true if the change in the theta direction is less than the maximum allowed change in the theta direction for doublets.

This process is continued until all hits in the data set have been used as inner hits.

3 Results

Four implementations of seeding algorithms were tested. Each implementation uses different data structures and python libraries but they follow the same outline covered in the previous section.

3.1 Loop Method

This is the original method used in the D-Wave study. The Loop Method is written in standard python and does not use any external libraries. The method utilizes six loops to iterate over all possible inner/outer hit combinations. The benefit of this approach is that each loop allows the algorithm to iterate layer-wise, phi-wise, and hit-wise. This means that the function can filter layers and phi slices without ever looking at the hits inside these regions. This cuts down on the average number of calculations needed per outer hit.

The algorithm was tested in a virtual machine on a laptop using two Intel Core i7 processors. The algorithm was run on 20 samples from the simulated data set. Each sample contains an increasing fraction of the hits expected in HL-LHC conditions starting from 5 percent of the HL-LHC data up to 100 percent, in 5 percent increments. A graph of the resulting runtimes are shown in Figure 6.

Then, the algorithm was tested on a CPU node with 64 threads on the NERSC Cori Haswell supercomputer. The algorithm was again run on twenty sample data set in 5 percent increments starting from 5 percent of the data up to 100 percent of the data set. A graph of the resulting runtimes is shown in Figure 7.

3.2 GUvectorize Method

This method loads all the hits into a large two-dimensional Numpy array. The method also uses the Numba python library [6]. The GUvectorize function decorator from the Numba library was used to convert the filter function from a standard python implementation into a Generalized Universal Numpy function. Generalized Universal Numpy functions allow for fast computation across Numpy arrays. Furthermore, the jit function decorator was applied to the function responsible for calculating the region of interest as well as other auxiliary functions. Finally, the nopython flag was set to true for all functions implemented with Numba. This allows the functions to be compiled at runtime and run as fast as code written in C.

The same tests that were run on the loop method were run on the GUvectorize method and are shown in Figures 6 and 7.

[6] https://numba.pydata.org/

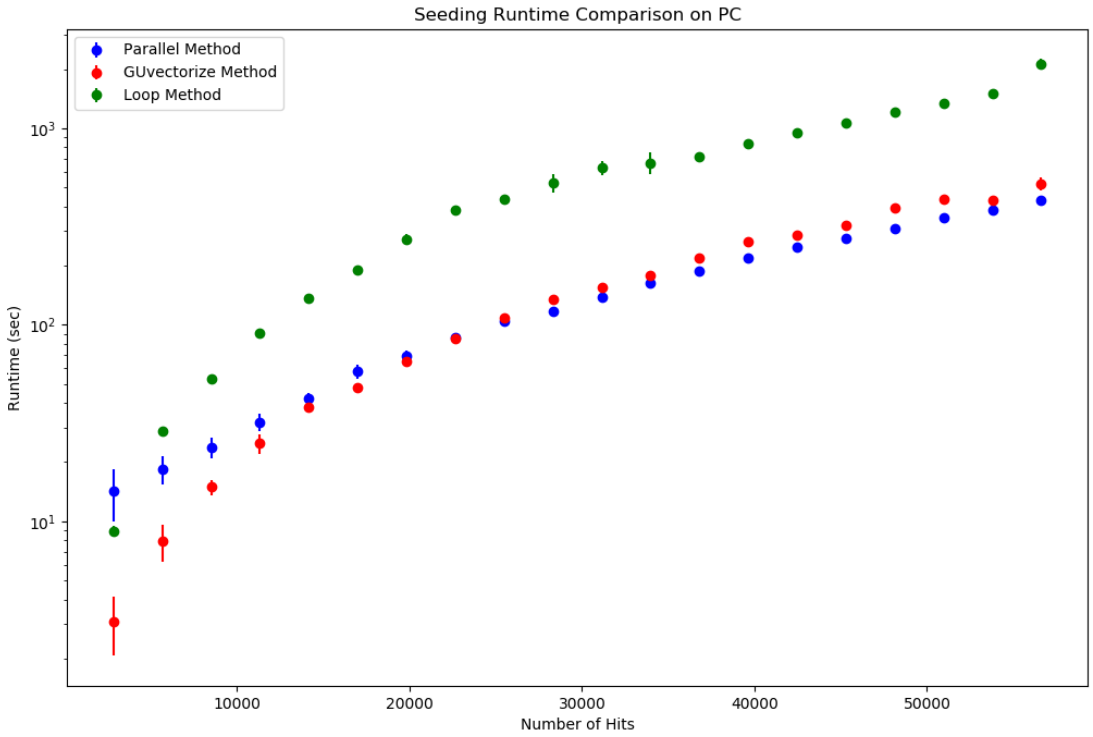


Figure 6: The performance of the seeding methods run on the same laptop and data set

3.3 Parallel Method

This method uses the same basic data structure as the GUvectorize method. The goal of this method is to run the inner hit loop in parallel. This is done by using the jit decorator on the inner hit loop and setting the parallel key word argument to true. Numba does not support calls to universal Numpy functions inside jit functions, so jit was used to compile the filter function instead of GUvectorize.

The results are shown in Figures 6 and 7.

3.4 Pandas Methods

This method is analogous to the Numpy method. However, this method replaces the two-dimensional Numpy array used in the Numpy method with a Pandas data frame. This algorithm performed worse than the Numpy method, taking 120 seconds to run on 5 percent of the data set. This is due to the way in which Pandas handles filtering of data frames. For each inner hit iteration, the data in the data frame is copied into a new data frame instance. The data frame contains a large amount of data and the inner hit iteration is executed for each row in the frame, so the combined time to copy this data frame adds up to a substantial amount of runtime. A second Pandas method was tried in an attempt to reduce this filtering problem as well as incorporate the layer and phi-wise iteration advantage that the Loop method benefits from. This was done by binning the z-direction such that the data frame could be multi-indexed so that the layer bin, phi bin, and z bin were each their own dimension in the data frame. This allowed for index slicing rather than the boolean indexing that the other methods used. However, this method still suffered from slow downs, taking 90 seconds to run on 5 percent of the data set.

4 Results and Discussion

On the complete data set, the parallel method ran about 4.68-times faster than the original loop method on a laptop and 24.56-times faster on Cori. The parallel method achieves the best performance by running the inner hit loop in parallel. The inner hit loop is a prime candidate

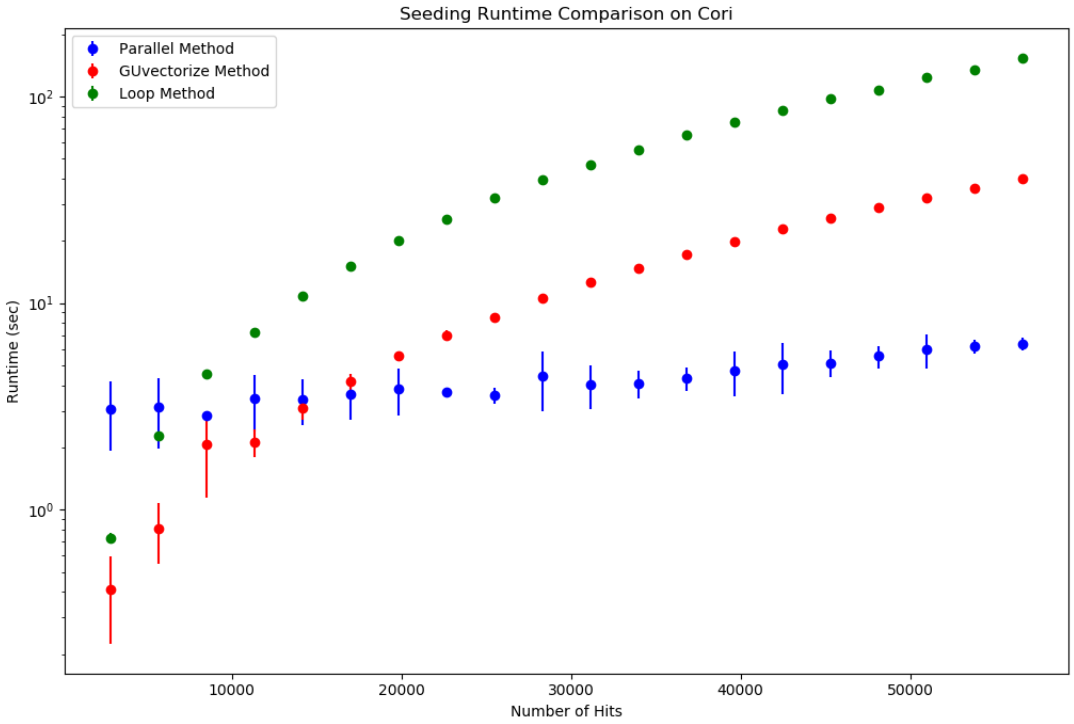


Figure 7: The performance of the seeding methods run on a Cori node

for parallelization for two reasons. First, the computations per iteration are complex enough to outweigh the latency time associated with running in parallel so gains in efficiency are possible. Each iteration through the inner hit loop takes the laptop configuration roughly three milliseconds to complete, on average. Second, each inner hit iteration does not depend on the others. In other words, the doublets which are formed using one inner hit are not changed by the doublets that are formed using a different inner hit. This allows for the loop to be run non-sequentially and achieves significant speed ups.

There is only a minimal difference in runtimes between the GUvectorize method and the parallel method when run on the laptop. This is because only two cores are being accessed by the virtual machine which limits the efficiency of parallelization. The speed ups that are being seen when the two algorithms are run on the laptop are most likely the result of the compilation of the code at runtime. And, the slight edge the parallel method gains over the GUvectorize method on the laptop is likely the result of the extra core being used.

The real difference between the two algorithm designs can be seen when they are run on Cori. On this hardware configuration, the parallel method has access to 64 hardware threads which maximizes the algorithm's capabilities. Here, the parallel method out performs the GUvectorize method by a significant margin even though both algorithms benefit from being compiled at runtime. This suggests the bulk of the parallel method's speed up derives from exploiting the additional threads.

Finally, the loop method is the slowest of the three seeding methods because it is not compiled at runtime and does not run in parallel. The method is limited because its large nested loop structure can not be run non-sequentially. The outcome of one loop is dependent on the other. This limits the algorithm to running its processes serially and does not take full advantage of the hardware configuration it is being run on.

5 Conclusion

The objective of the research was to improve the design of the seeding algorithm for annealing pattern recognition applied to charged particle track reconstruction. The python libraries Numpy and Numba were used to compile the code at runtime and run the computationally expensive loops in parallel. Four algorithm methods were developed and tested on a standard laptop configuration and on a supercomputing cluster. It was shown that the parallel seeding algorithm ran 4-times faster than the old seeding algorithm on the laptop configuration and 24-times faster on the super computing cluster. The future of the research will include modifying the parallel method with CuPy so that it can be tested on an NVIDIA GPU. The results of this study will improve the runtime of future annealing pattern recognition and machine learning studies applied to charged particle track reconstruction.

6 Acknowledgements

Thanks to Paolo Calafiura for his mentorship and guidance throughout my research. Thanks to Rollin Thomas and Laurie Stephey for their programming support. This work was prepared in partial fulfillment of the requirements of the Berkeley Lab Undergraduate Research (BLUR) Program, managed by Workforce Development and Education at Berkeley Lab.

7 Appendix: The seeding algorithm

This section demonstrates more explicitly the differences in the design for each algorithm. The Github repository is `https://github.com/sbconlon/Qallse_v2.0`.

7.1 Parallel Make Function

File location: `https://github.com/sbconlon/Qallse_v2.0/blob/Parallel_Numpy_Branch/../src/hepqpr/qallse/seeding/doublet_making.py`

```
98          @jit(nopython=True, parallel=True)
99          def make(approx_num_doublets=5000000):
100                 '''
101                 This function makes all possible doublets that fit the criteria of the filter. It first
102                 choses an inner hit and then iterates through the hit table looking for possible outer
103                 hit candidates. It then returns a list of hit ids cooresponding to the inner and outer
104                 hit pairs of the created doublets.
105                 '''
106                 ncolumns = int(nHits * 0.01)
107                 outer_2D = np.zeros((nHits, ncolumns), dtype=int64)
108
109                 for row_idx in prange(nHits):
110                         inner_hit = hit_table[row_idx]
111                         layer_range, z_ranges = get_valid_ranges(inner_hit)
112                         outer_hit_set = hit_table[filter(hit_table, inner_hit, layer_range, z_ranges)].T[0]
113                         for column_idx in prange(len(outer_hit_set)):
114                                 outer_2D[row_idx][column_idx] = outer_hit_set[column_idx]
115
116
117                 outer = np.reshape(outer_2D, (1, nHits * ncolumns))[0]
118                 inner = np.zeros(len(outer), dtype=int64)
119                 for row_count in prange(outer_2D.shape[0]):
120                         for col_count in prange(ncolumns):
121                                 inner[(row_count * ncolumns + col_count)] = hit_table[row_count][0]
122
123                 return inner, outer
```

Loops over all hits in the dataset using each one as the outer hit and only keeping those which meet the crieria

Loops over all hits in the dataset using each one as the inner hit

Figure 8: The areas of code highlighted in red indicate the three major loops that have been parallelized in the algorithm. The Numba keyword 'prange' manually parallelizes the for loop when the parallel parameter in the jit decorator is set to true. The areas of code highlighted in green show the Numpy functions that Numba parallelizes automatically, namely array construction and boolean indexing.

7.2 GUVectorize Make Function

File location: `https://github.com/sbconlon/Qallse_v2.0/blob/Numpy_Branch/src/hepqpr/.../qallse/seeding/doublet_making.py`

```
94          def make():
95                  '''
96                  This function makes all possible doublets that fit the criteria of the filter. It first choses an
97                  inner hit and then iterates through the hit table looking for possible outer hit candidates. It
98                  chooses two inner hits in an attempt to help balance the computation time for each loop.
99                  '''
100                 for indx in range(nHits//2):
101                         # If there are an even number of rows, the final iteration should be ignored
102                         if indx > (nHits-indx-1):
103                                 continue
104                         # If there are an odd number of rows, then the final iteration should only use one row
105                         if indx == (nHits-indx-1):
106                                 inner_hit = hit_table[indx]
107                                 layer_range, z_ranges = get_valid_ranges(inner_hit)
108                                 outerHitSet = np.array(filter(hit_table, inner_hit, layers_range, z_ranges))
109                                 for outerHit in outerHitSet:
110                                         doubletsStorage.inner.append(hit_table[indx][0])
111                                         doubletsStorage.outer.append(outerHit[0])
112                         # Otherwise, two rows should be used
113                         else:
114                                 inner_hit_one = hit_table[indx]
115                                 inner_hit_two = hit_table[nHits-indx-1]
116                                 layer_range_one, z_ranges_one = get_valid_ranges(inner_hit_one)
117                                 layer_range_two, z_ranges_two = get_valid_ranges(inner_hit_two)
118                                 outerHitSet_one = hit_table[filter(hit_table, inner_hit_one, layer_range_one, z_ranges_one)]
119                                 outerHitSet_two = hit_table[filter(hit_table, inner_hit_two, layer_range_two, z_ranges_two)]
120                                 for outer_hit_one in outerHitSet_one:
121                                         doubletsStorage.inner.append(inner_hit_one[0])
122                                         doubletsStorage.outer.append(outer_hit_one[0])
123                                 for outer_hit_two in outerHitSet_two:
124                                         doubletsStorage.inner.append(inner_hit_two[0])
125                                         doubletsStorage.outer.append(outer_hit_two[0])
126
127                 if debug:
128                         print(f'---> {len(doubletsStorage.inner)} Doublets Created')
```

This loop uses every hit as an inner hit. Each iteration uses two inner hits at a time which cuts the number of iterations in half

Each outer loop constructs the set of plausible outer hits for their respective inner hits.

Store the inner hit, outer hit pair ids in lists

Figure 9: This make function uses two inner hits per iteration. This was done in an attempt to balance the computation time per iteration because inner hits at the end of the dataset table have less outer hits to consider than the inner hits at the top. This due to the fact that a inner hit does not need to consider the hits that come before it in the hit table because that connection has already been considered. Also, this function stores the inner and outer hit IDs in a doublet storage class rather than returning them like the parallel method.

7.3 GUVectorize Filter Function

File location: `https://github.com/sbconlon/Qallse_v2.0/blob/Numpy_Branch/src/hepqpr/.../qallse/seeding/doublet_making.py`

```
@guvectorize([(int64[:, :], int64[:], int64[:], int64[:, :], boolean[:])], "(n, m),(m),(l),(l, o)->(n)", nopython=True)
def filter(table, inner_hit, layer_range, z_ranges, keep):
        '''
        This function combines the helper filters into one filter and is compiled by numba
        '''
        for row_idx in range(table.shape[0]):
                keep[row_idx] = (filter_layers(table[row_idx][1], layer_range) and
                        filter_phi(inner_hit[2], table[row_idx][2], nPhiSlices) and
                        filter_doublet_length(inner_hit[3], table[row_idx][3], minDoubletLength, maxDoubletLength) and
                        filter_horizontal_doublets(inner_hit[3], inner_hit[4], table[row_idx][3], table[row_idx][4], maxCtg) and
                        filter_z(table[row_idx][1], table[row_idx][4], layer_range, z_ranges))
```

Figure 10: This filter function uses the GUVectorize Numba decorator to compile the function into a generalized universal Numpy function.

7.4 Loop Method Function

File location: `https://github.com/sbconlon/Qallse_v2.0/blob/Original_Branch/src/...hepqpr/qallse/seeding/doublet_making.py`

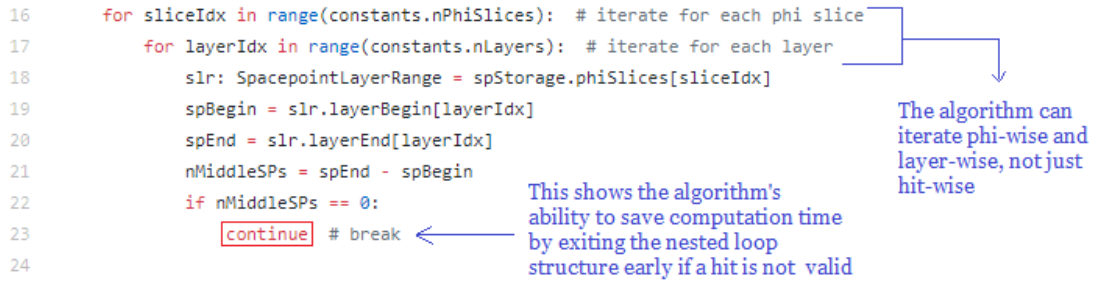


Figure 11: This section from the loop method's algorithm demonstrates the strengths of the code. The algorithm can iterate layer-wise, phi-wise, and hit-wise which allows for less computation because the algorithm can exit a loop at any time, saving further needless computation. Highlighted in red, the continue keyword is shown explicitly.

7.5 Pandas Data Frame Function

File location: `https://github.com/sbconlon/Qallse_v2.0/blob/Basic_DataFrame_Branch/...src/hepqpr/qallse/seeding/doublet_making.py`

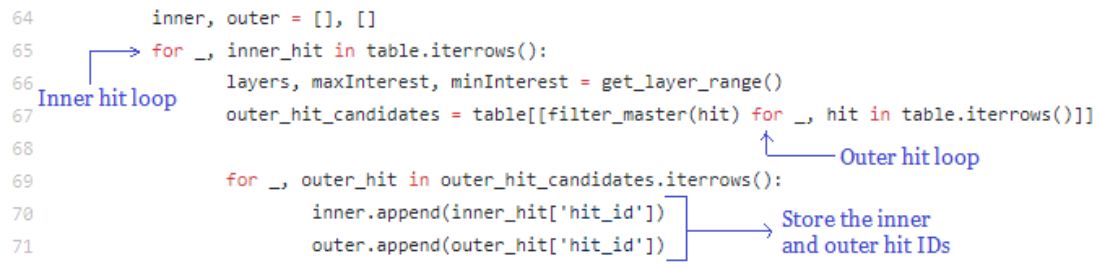


Figure 12: This is the simplest make algorithm of any of the methods. The algorithm iterates through all inner and outer hit combinations and only keeps those which meet the criteria of the filter master function.