FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**MASTER THESIS**

Tomáš Čelko

# Clustering hits and predictions in data from TimePix3 detectors

Department of Software and Computer Science Education

| | |
|---|---|
| Supervisor of the master thesis: | RNDr. František Mráz, CSc. |
| Study programme: | Computer Science |
| Study branch: | Artificial Intelligence |

Prague 2023

Firstly, I would like to express my gratitude towards my thesis advisor RNDr. František Mráz, CSc. from the Faculty of Mathematics and Physics, Charles University, for his patient guidance during the work on the thesis.

Additionally, I would like to thank MSc. Benedikt Ludwig Bergmann, PhD. from the Institute of Experimental and Applied Physics, Czech Technical University, for his ideas and help with the thesis.

A special thanks goes to Mgr. Ing. Petr Mánek, for the inspiration on the software design and also to all of my colleagues from the Institute of Experimental and Applied Physics for their insightful comments.

And last but not least, I am grateful to my girlfriend and family for their unending support.

Title: Clustering hits and predictions in data from Timepix3 detectors

Author: Tomáš Čelko

Department: Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc., Department of Software and Computer Science Education

Abstract:

Hybrid pixel detectors like Timepix3 and Timepix4 detect individual pixels hit by particles. For further analysis, individual hits from such sensors need to be grouped into spatially and temporally coinciding groups called clusters. While state-of-the-art Timepix3 detectors generate up to 80 Mio hits per second, the next generation, Timepix4, will provide data rates of up to 640 Mio hits, which is far beyond the current capabilities of the real-time clustering algorithms, processing at roughly 3 MHits/s.

We explore the options for accelerating the clustering process, focusing on its real-time application. We developed a tool that utilizes multicore CPUs to speed up the clustering. Despite the interdependence of different data subsets, we achieve a speed-up scaling with the number of used cores. Further, we exploited options to reduce the computational demands of the clustering by determining radiation field parameters from raw (unclustered) data features and automatically initiating further clustering if these data show signs of interesting events. This further accelerates the clustering while also reducing storage space requirements. The proposed methods were validated and benchmarked using real-world and simulated datasets.

Keywords: pixel detectors timepix3 clustering machine learning

# Contents

# Introduction

Over the years, many experimental physicists strove for a better understanding of particle interactions. The first step to comprehend the behavior of elementary particles is to observe them. That is also one of the purposes of the detectors developed in the Medipix collaboration facilitated by CERN. Even though the collaboration mainly focused on the medical application of the detectors, they are a valuable tool in the area of high-energy physics.

This collaboration produced a widely used detector chip called Timepix3 [1]. During the measurement, when an ionizing particle enters its sensor area, it causes the charge carriers to drift toward the sensitive pixels located in a $256\times256$ grid. The corresponding pixels then generate so-called hits. Timepix3 is capable of detecting these hits very precisely – with a pixel width of 55 $\mu$m and a time precision of 1.56 nanoseconds. Each particle (which may or may not decay into secondary particles) can generate multiple such hits, which together form a group known as a 'cluster' (created during 'clustering'). These clusters are then further analyzed by the physicists.

### Detector applications

Detectors of the Medixpix collaborations have been proven valuable for various applications in fundamental research [2] and life science [3]. Apart from that, there are other applications of these detectors worth mentioning:

- **Timepix3 – High-energy physics – particle identification.** With Timepix3, we can reconstruct the trajectories of the particles in the detected events and analyze their properties. Then, based on the recorded data, it is possible to infer which particles were likely interacting in the particular event. Additionally, it is possible to work with multiple detectors stacked on top of each other. Here, the task is to identify the tracks that coincide, which means they were likely generated by the same particle. This enables us to describe the trajectories in more detail and possibly improve the accuracy of particle identification (further described by [4]).

- **Timepix3 – Medicine – Ion beam radiotherapy.** In contrast to conventional radiotherapy, ion beam radiotherapy can better focus the beam at the target, which minimizes the potential damage caused by the beam to the surrounding healthy tissue. Even though this can be viewed as an advantage, it also introduces another challenge – the quality of the delivered dose distribution becomes more sensitive to the changes in the stopping power of patient tissue. By utilizing helium ion radiation and the Timepix3 detectors, it is possible to obtain high-contrast images of the beam profile, as shown in [5]. Such images are then used to improve the accuracy of the beam and minimize the collateral damage to the healthy tissue.

- **Timepix [6] (Timepix3 predecessor) – Radiation monitoring in Space – The Space Application of Timepix-based Radiation Monitor (SATRAM).** 'SATRAM is a spacecraft platform radiation monitor on board the Proba-V satellite launched in an 820 km altitude low Earth orbit in 2013.' [7] The goal of this detector is to determine the composition (particle type) and spectral characterization (energy loss) of mixed radiation fields around the Earth's orbit.

Currently, a new Timepix4 [8] chip is in development, with an even larger pixel matrix of 440×512 pixels and a temporal resolution of as few as 200 picoseconds.

It is evident that the capabilities of the new chips also present more challenges for the software. With the increase in data rates, there is a need to build efficient algorithms for data processing.

As one of the first and most essential steps of the data analysis is clustering, we explore possibilities to make the clustering faster. We aim to discuss and implement three options to increase clustering efficiency, as discussed below.

Parallel clustering. By effectively distributing the work between multiple CPU cores, we could increase the processing hit rate of the clustering. Unfortunately, the increase in speed is not guaranteed, as we can expect a non-trivial overhead from the parallelization.

Approximative clustering. Instead of producing the exact clusters, we might want to sacrifice the quality of clusters for a possible performance gain. For instance, we can utilize the observation that high-energy hits tend to be surrounded by other hits or relax the definition of the cluster.

Selective (machine-learned) clustering. Unsurprisingly, not all of the large amounts of data are considered 'interesting' for physicists. In principle, such data do not necessarily need to be clustered. In some cases, it might be difficult to exactly describe which data is worth clustering. Here, machine learning (ML) can be used to pick the 'interesting' data for clustering. As this decision needs to be made in a very short time, it is critical that the inference of the ML model is computed quickly, which limits the selection of viable ML models.

### Clustering applications

Standardly, the clustering is performed 'offline' after the raw data is acquired and stored on disk. An efficient clustering algorithm could allow clustering in real-time, even for radiation sources with high particle fluxes. And instead of saving raw data (hits), the clustered data can be saved to the disk.

Another advantage of real-time clustering is the possibility of subsequent real-time filtering of clusters. The capabilities of post-clustering filtering are much higher as the creation of clusters introduces many more measurable features (which are usable for filtering).

Additionally, the data reduction from selective clustering could allow the detectors to have a larger uptime and process more data while either discarding the redundant data or compressing it into short statistics.

### Thesis layout

In Chapter 1, we describe the Timepix3 detector and the data it produces. We also define the key terms and discuss the existing clustering approaches. In Chapter 2, we present the main goals this thesis aims to achieve. In Chapter 3, we discuss the options for the distributed computation of the clustering algorithm. In Chapter 4, an approximative clustering method is discussed, which tries to trade cluster quality for the potential acceleration of the clustering. In Chapter 5, we describe selective clustering, with an application of machine learning. In Chapter 6, we perform multiple experiments to analyze the correctness of the proposed methods. Furthermore, the maximal speed of the clustering methods is tested on various datasets.

# 1. Analysis

This chapter introduces the basic concepts relevant to clustering and summarizes the related work in the field. Additionally, we provide the definitions of the key terms used further in the thesis. First, in Section 1.1, we describe the origin, properties, and usage of the Timepix detectors. Second, in Section 1.2, we provide the basic properties of the Katherine – a frequently used piece of hardware to extract data from the Timepix detectors. Moreover, we also discuss the data it produces. Then, in Section 1.3, we define the clusters and the process of their creation, called clustering. We also discuss the existing clustering approaches.

## 1.1 Timepix detectors

### 1.1.1 The Medipix family

The origin of the Timepix detectors can be traced back to the 1990s. At that time, the first collaboration, Medipix1, was facilitated by CERN (European Organization for nuclear physics). The goal of the collaboration was to extend possible applications of the technology, primarily intended for the large hadron collider (LHC), to other areas, like medicine.

One of the results of the collaboration was the first Medipix1 chip. It was equipped with a 64x64 pixel sensor, where each pixel had a specified voltage threshold. The pixels would generate a hit if the induced charge at the pixel were higher than the threshold. In principle, it operated similarly to a standard camera – the detector counted hits while its shutter was open. This process generated image-like data.

A few years later, the follow-up Medipix2 collaboration started. It produced the Timepix chip, which had notable improvements when compared to the Medipix1. It came with a larger sensor with $256 \times 256$ pixels. The chip was programmable to operate in three modes. First, it could count the hits above the threshold. Second, it could record the time when the induced charge exceeded the threshold, called time over the threshold (ToT). And third, it measured the time of arrival (ToA), which is the moment when the threshold was exceeded. The programmability vastly extended the field of possible applications, which is why this chip is still used, more than twenty years after its creation.

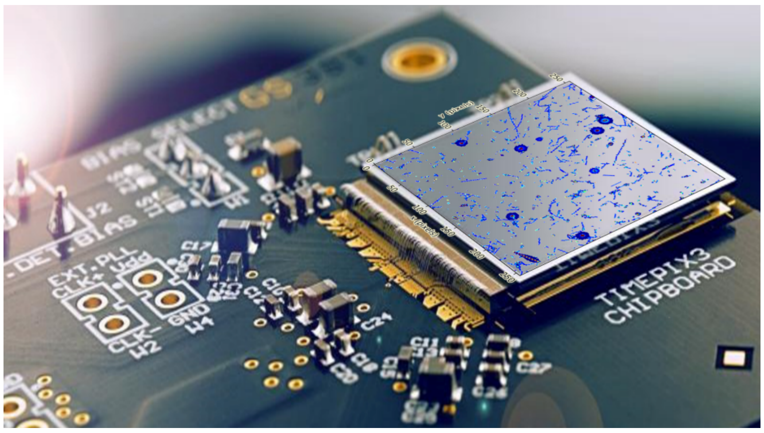


Figure 1.1: An image of Timepix3 detector with silicon sensor [9]. To better illustrate the working principle, the sensor has an overlay of sample data created by the detection of charged particles.

### 1.1.2 Timepix3

The third collaboration, Medipix3, provided two major improvements to its predecessor. In the new Timepix3, (for illustration, see Figure 1.1) chip's data-driven mode, the shutter was removed, and the frame-based processing was replaced by continuous sparse data readout. In continuous sparse data readout, the hits are transferred asynchronously in a stream. This way, the pixels with no hits do not need to be transferred, effectively reducing the amount of produced data and allowing faster further data processing. Moreover, it allows for simultaneous readout of both the time of arrival (ToA) and the time over the threshold (ToT). In the table below (Table 1.1), we present some of the characteristic features of the Timepix3 detector.

| Features of the Timepix3 detector | |
|---|---|
| Feature | Value |
| Pixel matrix | $256 \times 256$ |
| Pixel size | $55\ \mu m \times 55\ \mu m$ |
| ToA precision | 1.56 ns |
| Numbers of bits per hit | 48 |
| Data sent per hit | spatial coordinate (16 bits), ToT (10 bits), ToA (14 bits slow clock + 4 bits fast clock), and four extra bits (metadata with hexadecimal value 0xA) to mark the packet as data packet |
| Maximum data rate | 40 million hits per second per cm$^2$ |
| 'Slow clock' frequency | $f_{\text{slow}} = 40$ MHz |
| 'Fast clock' frequency | $f_{\text{fast}} = 640$ MHz |

Table 1.1: Basic properties of Timepix3 detector.

**Data acquisition**

The data acquisition with Timepix3 is a complex process that consists of multiple steps (see also Figure 1.2)

1. **Ionization.** When the charged particle traverses the sensor material, it creates ionization along its path. The ionization then results in the electrons moving from the valence band to the conduction band.

2. **Drift.** When the detector is powered on, there is a constant electric field in the area of the sensor. This field causes the charge carriers to drift toward the pixel electrodes in the sensor. Besides the electrons, the positively charged holes can be used as the charge carriers, which move in the opposite direction than the electrons. Let us denote the time of drift of a charge carrier $t_{\mathrm{drift}}$. The maximum possible value of $t_{\mathrm{drift}}$ depends on many physical properties of the measurement, like the strength of the electric field, the material of the sensor, the thickness of the sensor, and the temperature.

3. **Charge induction.** When the charge carrier approaches the pixel, it starts inducing its charge into a sensitive area of the pixel. This way, the signal is created.

4. **Signal amplification.** The amplitude of the signal induced in the pixel is too small for further processing. It is first amplified in the circuit near the pixel to enable correct measurement of the signal.

5. **Digitization.** After the amplification, the detector reads the signal from each pixel. In the readout electronics, the analog signal is digitized to the values corresponding to the intensity of the received analog signal. The digitization is performed for the prespecified signal threshold. The digitized values are then the aforementioned time over threshold and time of arrival.

6. **Read out and processing.** The digital signal is then read out by the user for further processing. In the data-driven mode, one of the first processing steps typically includes grouping the hits together to form an event, also known as clustering. Then, based on the properties of the clusters, the data is filtered, and last, the data is written to a disk for further 'offline' analysis.

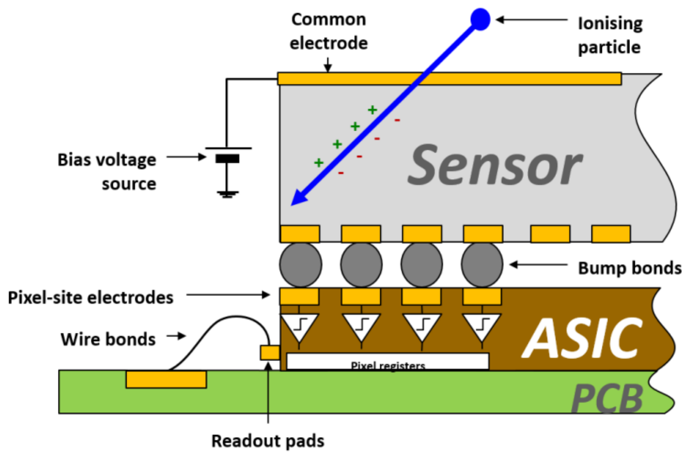


Figure 1.2: An illustration [9] of the data acquisition process with Timepix3, described by the part 1.1.2.

### Partial orderliness

On the chip, the data from the sensor are processed column-wise, which does not guarantee that the individual hits are temporally ordered. In other words, given hits $h_\mathrm{A}$ and $h_\mathrm{B}$ which occurred at times $t_\mathrm{A}$ and $t_\mathrm{B}$, $t_\mathrm{A} < t_\mathrm{B}$, the hit $h_\mathrm{A}$ could be transferred later than $h_\mathrm{B}$. Nevertheless, from the properties of the hardware, we can provide the bound on the 'orderliness' of the data. For that, we define the term $t$-ordered sequence.

Let $\{t_1, t_2, \ldots, t_n\}$ be a sequence of the time of arrival of hits received by the computer from the readout. We say this sequence is $t$-ordered if for all indices $i < j$ it holds $t_i < t_j + t$.

For the Katherine readout, it was assessed that for $t_\mathrm{max} = 500\mu s$, the hits are $t_\mathrm{max}$-ordered. It implies that after receiving the hit with the time of arrival $t_\mathrm{current}$, all the hits with the time $t < t_\mathrm{current} - t_\mathrm{max}$ must have already arrived (otherwise, it would be in a contradiction with $t$-orderliness). As we will see later, this observation simplifies the required sorting.

### Time measurement

For better accuracy of the measured time information, we utilize information from both the 'slow' and the 'fast' clock. The time of arrival in seconds can be computed given the number of ticks of the slow clock $ToA_\mathrm{slow}$ and the fast clock $ToA_\mathrm{fast}$ as $ToA = \frac{ToA_\mathrm{slow}}{f_\mathrm{slow}} - \frac{ToA_\mathrm{fast}}{f_\mathrm{fast}}$. This, however, is not sufficient to compute the true value of $ToA$. During the measurement, we can notice that the rise time of the signal for a pixel does not depend on the signal amplitude; see Figure 1.3. Here, the rise time is defined as the time from the start of the signal rise until it

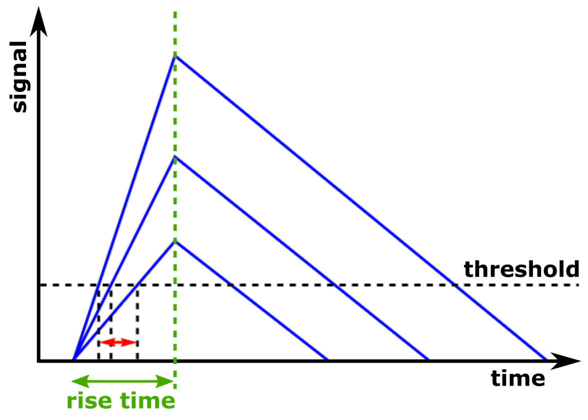


Figure 1.3: Dependence of time of arrival on the signal amplitude. The red arrow denotes the ToA differences.

reaches its peak. Therefore, we can see that signal with a lower amplitude reaches the threshold slower than the high-amplitude signal. Thus, we need to lower the value of ToA for low-amplitude signals. This phenomenon is called the 'Timewalk effect'. The parameters for the compensation are set during the calibration [10].

#### Energy measurement

Because of the independence of the rise time and the amplitude, the non-linearity is also observed in the measurement of the deposited energy. This time, however, we are measuring the time over the threshold, which means the compensation should be inverse compared to the compensation for ToA. Thus, the ToT needs to be increased for low-amplitude signals. Because of the energy calibration, we can convert the ToT values directly to the energy deposited in the sensor. The resulting conversion formula between the ToT value and deposited energy $e$ in the hit of a pixel $x$ is then shown below.

$$ToT(x) = a \cdot e(x) + b - \frac{c}{e(x) - t}$$

### 1.1.3 Timepix4

The newest member of the Medipix detector family was named Timepix4 (see Figure 1.4). As of now (2023), it is still under development, but the first prototypes have shown promising results. With the increased size of the sensor to $448 \times 512$ pixels and an even better time resolution of 200 picoseconds, it offers more detailed particle tracking. On the other hand, because of its capability to handle extreme particle fluxes of up to 3.4 MHz per mm$^2$ per second, it can produce huge amounts of data, which puts more pressure on developing fast and efficient data processing methods.

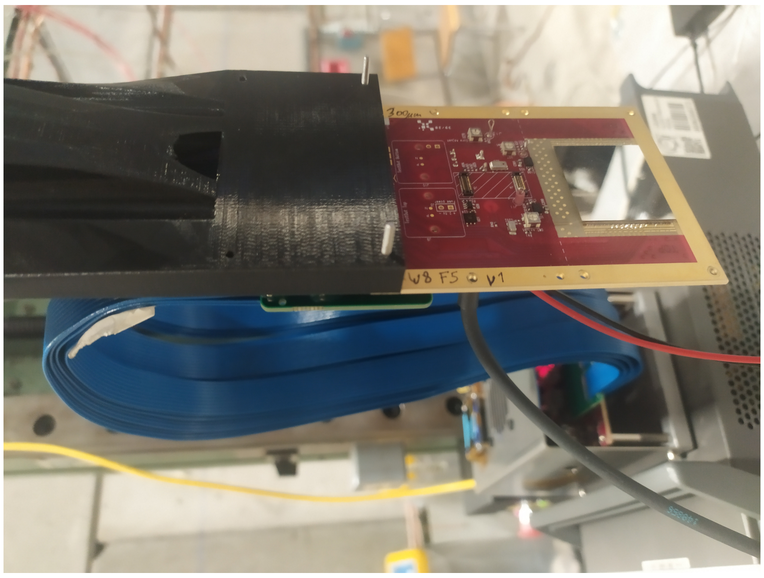


Figure 1.4: A photo of Timepix4 taken during a measurement.

## 1.2 Katherine readout

The Katherine readout (see Figure 1.5) is a device that handles the communication between the Timepix3 chip and a computer. The main advantage of this readout is that it supports a long-distance Ethernet connection. The cable length can reach up to 100 meters, which is especially useful in environments where human interference is impossible. Such a scenario is rather common during measurements with highly ionizing radiation. Depending on the used cable types, the maximum hit rate supported by Katherine is as high as 16 MHit$\cdot$s$^{-1}$. Notably, longer cables decrease the communication speed between the readout and the detector.

Figure 1.5: An image of the Katherine readout device [11]

## 1.3 From hits to clusters

The hits produced by the detector carry three pieces of information – spatial, temporal, and information on induced charge ($\approx$ energy). Formally, the hit can be defined as a tuple $(x, y, ToA, ToT)$, where $x$ and $y$ are the pixel coordinates, $ToA$ is the time of arrival and $ToT$ is the time over the threshold. However, it is difficult to draw any conclusions from the data contained in a single hit. Additionally, the hits can be seen only as a feature describing an event near the sensor. Each event can be mapped in a $1 : n$ relationship to the hits detected by the sensor. To define this mapping, we need first to define a few terms concerning the neighborhood of a hit. We say two hits with spatial coordinates $(x_1, y_1)$ and $(x_2, y_2)$ are spatially neighboring if $|x_1 - x_2| \leq 1$ and $|y_1 - y_2| \leq 1$. This type of neighboring is often referred to as an 8-fold neighborhood. Similarly, two hits with the times of arrival $t_1$ and $t_2$ are considered temporally neighboring for a predefined constant $dt_{\max}$, if $|t_1 - t_2| \leq dt_{\max}$. Utilizing these two definitions of the neighborhood, we can define the cluster of hits.

Suppose that $C = \{h_1, h_2, \ldots, h_n\} \subset H$ is a subset of all received hits $H$. We say that $C$ forms a cluster if, for each pair of hits, $(h_i, h_j)$ in $C$, the following conditions hold:

(i) There exists a path $P = a_1, a_2, \ldots, a_m$, where $a_1, a_2, \ldots, a_m \in C$. Additionally, $a_1 = h_i$ and $a_m = h_j$ and for all $k \in \{1, 2, \ldots, m-1\}$ the hits $a_k$ and $a_{k+1}$ are **both temporally and spatially neighboring**. Therefore $P$ is the path of neighboring pixels connecting $h_i$ and $h_j$.

(ii) There does not exist a hit $h \in H \setminus C$ both temporally and spatially neighboring with the hits in $C$. This condition corresponds to the fact that we are searching for all hits that can be added to a cluster.

The motivation behind this definition of the cluster stems from the fact that the particle trajectory is modeled as a continuous curve in time and space. After discretization into pixels, the continuity assumption is naturally replaced by the assumption of the spatial and temporal neighborhood of the pixels.

We will consider the aforementioned definition of a cluster to be the main definition of a cluster if not stated otherwise. Nevertheless, there also exist other definitions of a cluster. For instance, a different definition can be called the 'fixed-time-window cluster', which modifies the standard cluster conditions by introducing a new constraint: Each pair of pixels in a cluster must be temporally neighboring. By using this constraint, we restrict ourselves to only finding clusters with a fixed timespan of at most $dt_{\max}$.

## 1.4 Cluster morphology

During the measurements, different particle types might produce different clusters. These clusters can be divided into categories based on their shape. The most common cluster types are displayed in Table 1.2.

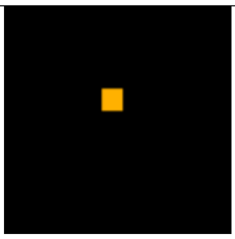
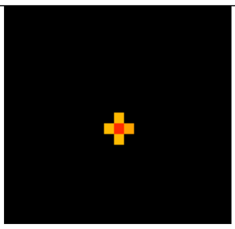
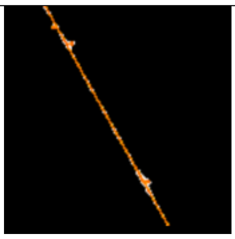
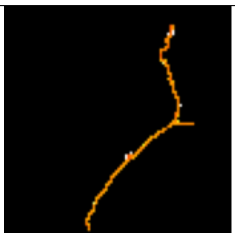
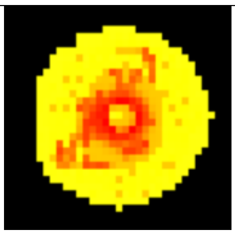
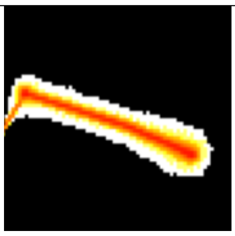
| Cluster types | | |
|---|---|---|
| Cluster category | Description | Prototype |
| dot | Dot clusters are single-pixel clusters. The source of such tracks can be low-energy photons and electrons traversing the detector almost perpendicularly to the plane of the sensor. |  |
| small blob | Compared to the dots, small blobs consist of more pixels. They are usually caused by X-ray photons and electrons. |  |
| straight track | Straight track is a trajectory with a linear shape, likely produced by energetic light particles traversing the detector under a greater (non-perpendicular) angle. |  |
| curly track | Curly track clusters are characterized by tracks with a non-linear shape of the branched curve. They are often produced by charged particles like higher energy electrons or gamma rays. |  |
| heavy blob | While being larger than a small blob, they are typically produced by highly-ionizing particles traversing the sensor orthogonally (at a low angle). |  |
| heavy track | Heavy tracks mostly have a linear shape. In contrast to straight tracks, they are produced by particles with higher energy, generating hits in many more pixels along their trajectory. |  |

Table 1.2: Cluster categories based on their shape.

Notably, the categories do not provide complete coverage of all possible clusters. The so-called 'exotic' tracks might not belong to any of these categories and require further analysis.

## 1.5 Existing clustering approaches

The clustering, as we defined it earlier in this chapter, is an essential step in the data analysis chain. Let us discuss the clustering methods used in practice.

### 1.5.1 Frame based clustering

For detectors that function in a shutter-based mode (like Timepix), the output data can be viewed as a sequence of image frames. Each such frame would then be processed separately – we can analyze it like a graph $G = (V, E)$. For each hit in the frame, we add a node to $V$. Then, if hits are neighboring (both spatially and temporally), we add a corresponding edge between them to $E$. Here, the task of clustering can be translated to the task of finding connected components in a graph, which can be solved by a graph search algorithm. For instance, depth-first search or breadth-first search can be used to identify each component and create clusters. Luckily, the graph does not need to be constructed in advance. We can utilize the 2D matrix representation and search through 8-neighbors. Here, the existence of the shutter comes with its drawbacks. For instance, the clusters, which are separated only in temporal dimension and are present in the same frame, would be incorrectly seen as a single cluster. Conversely, a cluster that occurs near the closing time of the shutter can be incorrectly split into two or more clusters.

### 1.5.2 Data-driven clustering

In the case of the data-driven output mode of a detector, it is possible to employ a more 'online' approach, which would process the data pixel-wise. In principle, both algorithms proposed below handle three possible scenarios based on the set of clusters to which the hit can be added. If there is no such cluster, we create a new cluster. If the hit can be added only to a single cluster, we simply add the hit to the cluster. Otherwise, if there are multiple clusters, we first need to merge them and then add the hit. The difference between these approaches lies in the method of finding the neighboring clusters for received hits and also in the clusters they produce.

1. **Quad-tree clustering.** We could use an algorithm described in the master thesis of L. Meduna [12] and [13]. A slightly modified pseudocode of this method is shown in Algorithm 1.

**Algorithm 1** Data-driven quadtree clustering [12]

```
procedure PROCESSHIT($hit$)
    $openClusters \leftarrow \emptyset$
    $added \leftarrow$ false
    for $cluster$ in $openClusters$ do
        if canBeAdded($cluster, pixel$) then
            if $added$ then
                $lastCluster \leftarrow$ mergeClusters($lastCluster, cluster$)
            else
                addHit($cluster, hit$)
                $added \leftarrow$ true
                $lastCluster \leftarrow cluster$
            end if
        end if
    end for
    if not $added$ then
        $lastCluster \leftarrow$ createNewCluster($openClusters$)
        addHit($lastCluster, hit$)
    end if
    closeAndDispatchOldlClusters()
end procedure
```

For this algorithm, we still need to explain how to check if a hit can be added to a cluster. In the Algorithm 1, this check is performed in the function 'canBeAdded'. This is the place where quadtree comes into play. So let us define the quadtree [14]. Suppose we are given a fixed-size bitmap with pixels having the value zero or one. Quadtree is a tree-like data structure with the following features:

(a) Each node corresponds to a tile of pixels.

(b) The tree consists of two types of nodes – internal and external. Internal nodes have four children nodes, while external nodes have none. External nodes correspond to tiles that are homogenously occupied by zeros or ones. Internal nodes correspond to the tiles not fully occupied by a single value and are therefore split further until the whole tile is occupied by either zero or one.

(c) The most frequent division into tiles is the division with a pair of orthogonal lines parallel to the sides of the bitmap image. Then, each sub-region is divided recursively. For the bitmap of size $2^n \times 2^n$, the tree can reach the maximum depth of $n$.

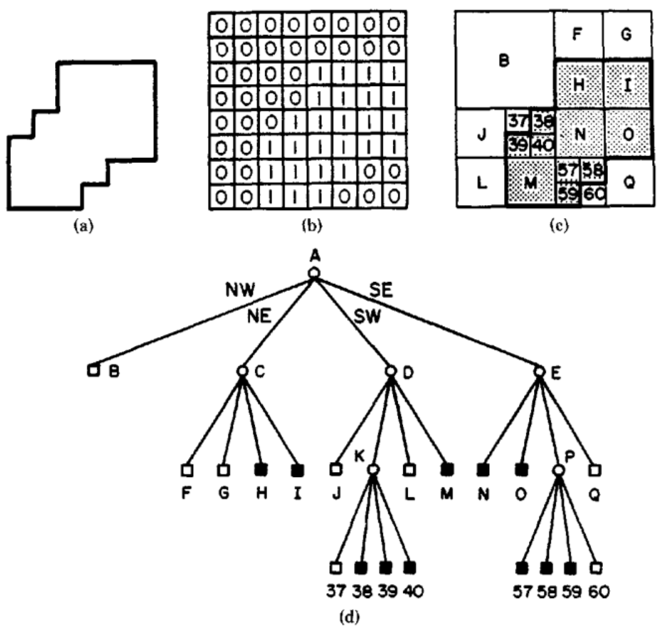


Figure 1.6: (a) Area homogeneously occupied by ones (in our case they are hits). (b) Pixel matrix. (c) Names of the tiles used in the quadtree. (d) The created quadtree labeled by the corresponding tiles. [14]

Quadtrees are an efficient compression method for image data. For the tiles filled with the same value, only a single node is created, effectively saving space compared to the naive implementation. The effectivity of the compression depends on the specific image. It is least effective for the 'chessboard' pattern. The bigger the areas with the same value pixels, the better the compression. For an example of the quadtree, see Figure 1.6.

For our application, we can use the quadtree to store the neighbor pixels of a cluster. By neighboring pixels for each cluster, we mean all of its pixels and their eight neighbors. Then, the 'canBeAdded' check can be translated to the 'find' operation in the quadtree. This operation has the expected time complexity $O(\log n)$, where $n$ is the size of the cluster. After each modification of the cluster, the quad-tree is modified to contain neighbors of a newly added pixel.

It is worth noting that the 'canBeAdded' method handles only spatial neighborhood condition, while the temporal neighborhood is partially handled by periodically dispatching the old clusters (see line 18 of Algorithm 1). An attentive reader might notice that this approach does not align with our definition of a cluster but rather the fixed-time-window cluster, which we consider to be a potential drawback of this algorithm.

2. **Pixel-list clustering.** Compared to quadtree clustering, this approach

uses a different implementation of 'canBeAdded' method. It was proposed in the master thesis of P. Mánek [15]. Instead of an auxiliary quadtree to store potential neighbors, at the beginning of the clustering, we create a matrix with an entry for each pixel. Each field in the matrix contains references to all currently open clusters containing the particular pixel and also the reference to the corresponding hits. Upon registering the hit, all of its eight neighbors are scanned in the matrix to find all of the neighboring hits and their clusters efficiently. The time complexity of the 'canBeAdded' check is thus constant with respect to the number of pixels. In contrast to quadtree clustering, the reference to the actual hit enables us to use also temporal information for neighborhood check. We can see that this approach complies with our definition of a cluster. Therefore, we decided to reimplement this algorithm and use it as the baseline clustering algorithm.

# 2. Goals of the thesis

In this chapter, we will briefly discuss the main objectives of the thesis. The thesis aims to:

- **Analyze existing approaches used for clustering and their properties.** This step is required to better understand the clustering and be able to build on the existing algorithms.
- **Design algorithms for distributed computation of clustering**. This enables the parallelization of the clustering, possibly leading to faster clustering.
- **Propose methods for approximative clustering.** The motivation here is to trade the quality of the clusters for the possible increase in the speed (throughput) of clustering.
- **Propose methods for selective clustering.** By selective clustering, we mean clustering only parts of the data which are specified by the user. The data of interest should be specified by the user in one of two ways:
  - **Explictly.** User should be able to define the data of interest by explicitly listing the desired features of the data stream.
  - **Implicitly (machine-learned).** The data of interest could also be selected implicitly by specifying existing examples of data with a positive or negative label. This would allow the application of machine-learning-based models which would aim to generalize which data is worth clustering. A simple GUI for the training of such models should be implemented.

  An application capable of both kinds of selective clustering based on user parameters should be implemented. Additionally, the application should provide visualization of these features, helping the user to decide which data should be selected and then classified into one of the two classes (either worth clustering or not worth clustering).
- **Ensure that proposed methods are suitable for real-time clustering applications.** Real-time data clustering is desired as it expands the possibilities of online data filtering. (Filtering on features of a cluster can be much more sensitive and selective than filtering individual hits). This data reduction could save a significant amount of storage space by discarding 'uninteresting' clusters in real-time.

- **Verify correctness of proposed approaches.** The verification should be performed by comparison with some of the existing clustering approaches on the measured data.
- **Evaluate the performance of the implemented clustering and machine learning models.** It is critical that the performance is evaluated using multiple datasets with various types of clusters. This ensures the clustering algorithms are suitable for clusters of many shapes and sizes.

# 3. Clustering parallelization

Before we start with the clustering acceleration, let us first introduce the motivation behind the speedup of clustering. By clustering, we mean partitioning hits received from the detector into groups based on temporal and spatial similarity, as defined in Section 1.3. The Timepix3 detectors can produce large amount of data that need to be processed. Such processing consists of multiple steps, where the clustering part is one of the most computationally expensive. Successfully accelerating this process would simply shorten waiting times for the experimental physicists while also enabling real-time data processing of radiation sources with higher frequencies of ionizing particles.

In this chapter, we will discuss the options for accelerating the clustering process and analyze the properties of these approaches. First, we will identify the steps of clustering algorithms that can be performed independently, which is summarized in Section 3.1. Then, in Section 3.2, we propose methods of data splitting that enable the utilization of even more distributed computational power. The process of reconstructing the clusters affected by the split is shown in Section 3.3. And in Section 3.4, we combine the discussed methods into a complete 'computation architecture', which describes the complete dataflow between the independent parts of the algorithm, implemented as a producer-consumer design pattern [16].

## 3.1 Step-based parallelization

The step-based parallelization is the process where we split the clustering algorithm into multiple steps that can be performed independently in a pipeline. The process itself depends vastly on the specific clustering algorithm. In our case, we used 'pixel-list clustering', described in Section 2. It is worth noting that apart from potential speed-up, this approach introduces some degree of modularity to the algorithm, making it easier to customize or extend the clustering. The clustering algorithm can be divided into the following steps:

1. Input reading. The goal of this step is rather self-explanatory. Given the input file or input detector readout, read the hits and prepare them for further processing. This step also allows simple data manipulation (modify the true frequency of hits, . . . ) providing us the freedom to adjust the data source for benchmarking purposes.

2. Hit calibration. This step converts the hits from the raw format of Katherine's readout to the so-called 'MM hit' format including information about

the energy of the hit, which is usually more convenient for potential further analysis. Additionally, this step converts the time of arrival and time over the threshold in clock ticks to nanoseconds.

3. Time sorting. As the data from the Katherine readout is only $t$-ordered (see Section 1.1.2), additional sorting is required to obtain a fully ordered sequence. Notably, one might wonder if sorting is truly required. First of all, a lot of physicists consider it very practical to have the clusters fully sorted after the clustering. And even if we decide to resign on this condition, when we compare the expected timespan of a cluster $t_c$ (the timespan of a cluster is the time difference between the first and the last hit in the cluster), and $t$, which defines the $t$-orderliness, it usually holds $t \gg t_c$. This implies that we would need to keep the cluster in memory not for $t_c$ but for $t$ time units. This would effectively make assigning a new hit to the cluster more difficult, as more clusters would be 'open' at a time.

4. Clustering. In this step, the sorted hits are grouped into clusters. Here, pixel-list clustering is considered to be the baseline clustering algorithm.

5. Cluster writing. In this part, we write out the created clusters into files. This step can be preceded by filtering or analyzing nodes to reduce the number of clusters that are outputted. Notably, filtering can be done even before the clustering but only in a limited manner. Introducing the concept of a cluster creates many more options on which cluster attributes can be used for filtering. For example, in contrast to pre-clustering filtering, we can now estimate the shape of the cluster or even try to identify particles in the cluster. Alternatively, because of the online clustering, we can extract the cluster features and store only those – reducing the data size even further.

## 3.2 Data-based parallelization

Data-based parallelization is a concept that can be considered 'orthogonal' to the previous step-based parallelization. Instead of splitting the algorithm into steps, we now split the data into blocks and assign each block to some computing worker.

Let us start with a motivation example first. We are given a large dataset of non-clustered hits. Based on the properties of the detector, we can assume the dataset is $t$-ordered in time. We would like to cluster the data as fast as possible. Provided, we are given enough computational power, a valid approach to speed up the processing could be rather simple:

1. split the data into chunks,
2. cluster data in each chunk independently, and
3. handle the clusters at the border that might have been (incorrectly) split into different chunks.

However, there are still a few points that we need to address first.

- How do we split the data into chunks? Clustering is the process based on temporal and spatial similarities of the hits, giving us two natural options for splitting the data – spatially and temporally. In subsections 3.2.1 and 3.2.2, we will discuss these options in detail.
- How do we merge the clusters at the border? The answer to this question depends on the approach we used to split the data. However, it is crucial that this process is fast and efficient. If merging of border clusters takes more time than the clustering itself, then the whole parallelization approach would be no better than simple clustering. In that case, the merging process would become the new bottleneck of the algorithm.

  Another (and a very natural) way how to reduce the possible merging can be to increase the size of the chunk. The bigger chunks would imply fewer border clusters which seems ideal. Nevertheless, to see why increasing chunk size might not always be viable, let us modify our original setting. Instead of storing the data offline on the disc, the hits are received online from the detector in real-time. When the hits arrive online, we need to ensure that the work is split between the workers as evenly as possible at any given point in time. Using a large chunk size in real-time clustering could make the split uneven, making the parallelization ineffective. In the subsections 3.2.1 and 3.2.2, we describe this problem in detail.

### 3.2.1 Spatial parallelization

One of the data-based split options is dividing the data along the spatial dimension. Usually, the sensor of the detector resembles a matrix. Therefore, in the spatial dimension, we expect to obtain two coordinates – $x$ and $y$. We aim to split the 2D matrix in such a way that there are as few clusters at the border as possible and that the division of work between workers is as even as possible – under the assumption that every part of the sensor has an equal chance of being hit. Such an assumption might not be valid ‘locally’ (for a short time window, we are more likely to obtain hits close together). However, with high hit rates, and uniform distribution of radiation above the sensor, we can expect an even distribution of work. For example, given four workers, a natural choice for a split can be dividing the sensor into four equal quadrants.

Let us now analyze the properties of a spatial split.

- **Distribution of work.** We could expect equal distribution of work, as the sensor parts have the same area. This is valid only under the assumption of an equal particle hit rate distribution in the 2D plane of the sensor. If the radiation source creates a 'narrow beam' with the mean of the particle distribution far from the center of the sensor, the distribution of work may become skewed. In some scenarios, this uneven distribution of workload can slow down the clustering significantly.

- **Split condition.** Given the number of available workers, the aim is to split the area of the sensor evenly into $n$ parts. Furthermore, such a split should consist of rectangles that are as close to a square as possible, as we would like to minimize the total length of the border (as the square maximizes the area-to-perimeter ratio). This seems rather non-trivial for an arbitrary $n$, potentially requiring additional computational power only to compute the target clustering worker.

- **Expected number of border clusters.** To provide an estimate on the portion of clusters lying at the border, we make additional simplifying assumptions. First, we assume that a cluster is a ball with a diameter of $d$ pixels. Then, we count all positions where the center of gravity of the cluster is not further than $d/2$ pixels from the border. Considering splits, where all borders are parallel with a side of the sensor, we can estimate the number of these positions of the center as: $n_{\text{border\_cl}} = (l_1 \cdot d \cdot n_1) + (l_2 \cdot d \cdot n_2) - (n_1 \cdot n_2 \cdot d^2)$. Here $l_i$ is the length of the sensor corresponding to $i$-th axis, $n_i$ is the number of splitting lines parallel with $i$-th axis and $0 < d < min(l_1/n_1, l_2/n_2)$. The term $-(n_1 \cdot n_2 \cdot d^2)$ subtracts all areas where the border lines intersect, which were counted twice. For a split of the Timepix3 sensor into four equal quadrants, we obtain $256d + 256d - d^2$. Setting $d = 5$, we get 2535 pixels, which is approximately 4% of all pixels.

  To compute the expected number of divided clusters, we proceed analogically to the case with the expected number of border clusters. Naturally, all divided clusters are also border clusters. We only need to subtract those clusters which ended exactly on the border. With the simplifying assumption of all clusters being spheres with diameter $d$, we obtain $n_{\text{divided\_cl}} = (l_1 \cdot (d-2) \cdot n_1) + (l_2 \cdot (d-2) \cdot n_2) - (n_1 \cdot n_2 \cdot (d-2)^2)$. For the Timepix3 sensor and $d = 5$, we obtain 1530 pixels, which represents approximately 2% of all pixels. This means that even if the mean diameter of clusters is as small as $d = 5$, more than half of the border clusters are 'incomplete' and need merging. Furthermore, depending on the angle of traversing particles, the tracks can have tilted shapes with lengths higher than our choice of $d$, further increasing the fraction of bordering clusters in the data. Our sample choice of $d$ is also supported by real-world data, see Figure 3.1.

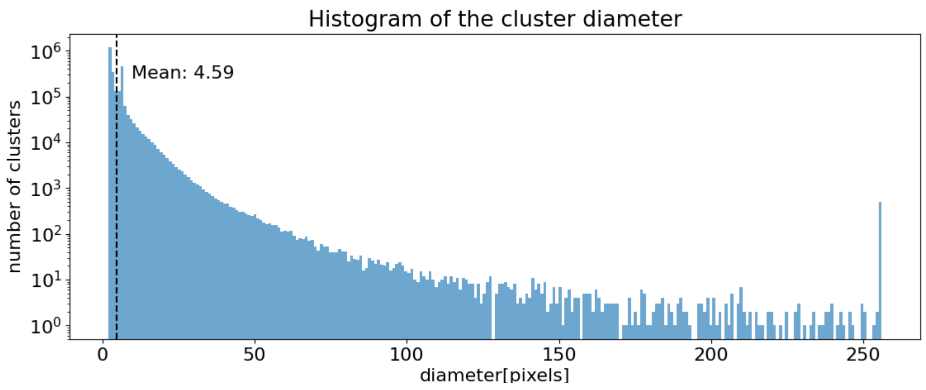


Figure 3.1: The histogram of the cluster diameter in the dataset consisting of various types of clusters from pion, lead ion, and neutron data we use further in the thesis experiments.

### 3.2.2 Temporal parallelization

Another option for dividing our data is division along the time axis. In contrast to the spatial dimension, the temporal dimension can be potentially unbounded. Therefore, each temporal window has to be mapped to one of $n_{\text{workers}}$ workers. One of such maps that assign hit with time $t$ to the worker with index $i(t) \in \{0, 1, \dots n_{\text{workers}} - 1\}$ can be $i(t) = (t\, mod(t_{\text{window}} \cdot n_{\text{workers}})\, div\, t_{\text{window}}$, with $div$ being integral division and $t_{\text{window}}$ being carefully chosen constant.

Let us discuss the properties of the temporal split:

- **Distribution of work.** Similarly to spatial parallelization, we can expect an even distribution of work if we assume that in every time window of size $t_{\text{window}} \cdot n_{\text{workers}}$, the pixel hits are uniformly distributed.
- **Split condition.** On contrary to spatial split, temporal split scales well for any number of workers, assuming $t \gg t_{\text{window}}$. We only need to update $n_{\text{workers}}$, and possibly $t_{\text{window}}$, but no other changes are necessary.
- **Expected number of border clusters.** To determine the number of border clusters, we first need to define the border. We say time $t$ is a border

time given constant $t_{\text{window}}$, if $t \mod t_{\text{window}} = 0$. During the clustering, we defined the constant $dt_{\text{max}}$ as the maximum time difference of hits to be considered temporally neighboring. Hence, we say the cluster is at the border if some of its hits are at most $dt_{\text{max}}$ distant from some border time. Such a definition was chosen for a simple reason – if the cluster is not at the border, it will certainly be processed by a single worker. On the other hand, If the cluster has a hit that is very close to the border (closer than $dt_{\text{max}}$), then it could happen that there might exist a hit on the other side of the border which should be part of the same cluster. Consequently, the probability that a hit falls onto the border can be computed as $p_{\text{border\_hit}} = \frac{2 \cdot dt_{\text{max}}}{t_{\text{window}}}$. Given that the average timespan of the cluster is $dt_{\text{cluster}}$, we can extend the last formula to estimate the probability of receiving a border cluster as $p_{\text{border\_cluster}} = \frac{2 \cdot (dt_{\text{max}} + \frac{dt_{\text{cluster}}}{2}))}{t_{\text{window}}}$. For $dt_{\text{max}} = 200$ ns, $dt_{\text{cluster}} = 25$ ns and $t_{\text{window}} = 10000$ ns we obtain the cluster border probability of approximately 2.5%.

The probability that the cluster was actually divided into separate windows can be estimated as the fraction: $p_{\text{divided\_cluster}} = \frac{dt_{\text{cluster}}}{t_{\text{window}}}$. Typically $dt_{\text{max}} > dt_{\text{cluster}}$, which implies that most of the border clusters are not split. For $dt_{\text{cluster}} = 25$ ns and $t_{\text{window}} = 10000$ ns we get $p_{\text{divided\_cluster}} = 0.0025$. Thus we can observe that under mentioned circumstances, the probability of dividing the cluster along the temporal dimension can be notably lower than the probability of dividing the cluster along the spatial dimension. Our sample choice of $dt_{\text{cluster}}$ is also supported by real-world data; see Figure 3.1.

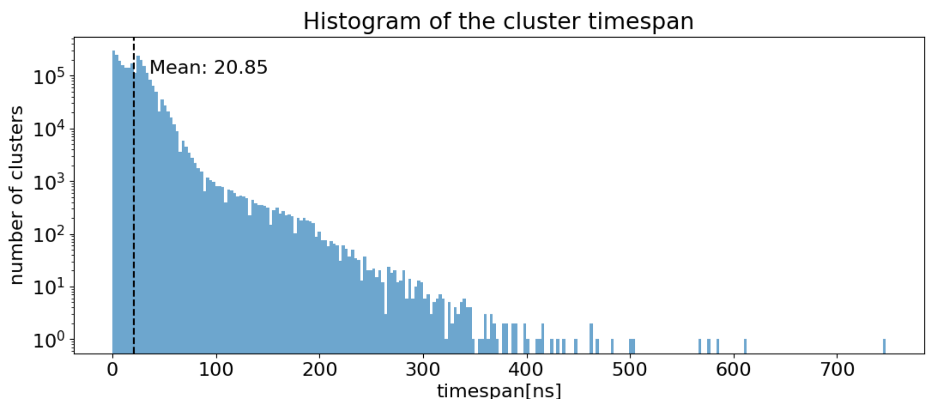


Figure 3.2: The histogram of the cluster timespan ($dt_{\text{cluster}}$) in the dataset consisting of various types of clusters from measured pion, lead, and neutron data.

### The choice of $t_{\text{window}}$

The $t_{\text{window}}$ parameter in the algorithm can be viewed as the parameter that balances two quantities:

- The number of potentially split clusters. As discussed in the previous part, the higher the value of $t_{\text{window}}$, the lower the number of potentially split clusters.

- Parallelization effectiveness. Even though the high values of $t_{\text{window}}$ imply less work in the merging part, it can also decrease the degree of parallelization by creating an uneven workload. In other words, if $t_{\text{window}}$ is large, and in the case of radiation sources with very high particle frequencies, we will have a huge number of hits in each window. Thus, it could happen that by the time the hits from a single window are correctly assigned by splitting the worker to the clustering worker, some of the other clustering workers could have already processed all of their hits and become idle, wasting computational resources. With lower $t_{\text{window}}$, the risk of such a scenario decreases.

One might wonder how to set $t_{\text{window}}$ correctly, keeping both parallelization effectivity and the number of divided clusters in mind. A simple way is to tune the $t_{\text{window}}$ parameter experimentally. Therefore, we need to run the clustering multiple times for each tested value of $t_{\text{window}}$ and choose accordingly.

### Avoiding the fixed $t_{\text{window}}$

Another option that may sound reasonable is not to split the data by some fixed time point but to draw the splitting line after a predefined number of hits – let us denote it as $h_{\text{wind}}$. As a consequence, we would have better control over the split of the work between the workers. On the other hand, splitting after a fixed number of $h_{\text{wind}}$ hits would lead to an increase in the number of split clusters. For instance, let us denote the average size of the cluster as $c_{\text{size}}$. Then the expected number of clusters in the window that are cut by a single border is at least $\frac{c_{\text{size}}-1}{c_{\text{size}}}$. Moreover, the number of all clusters in this window can be estimated as $\frac{h_{\text{wind}}}{c_{\text{size}}}$. Then, the probability of splitting the cluster is equal to the ratio of the mentioned expressions, which simplifies to $\frac{c_{\text{size}}-1}{h_{\text{wind}}}$. We can assume the $h_{\text{wind}}$ to be set according to the expected hit frequency $f$ as $h_{\text{wind}} = f \cdot t_{\text{window}}$. Consequently, in this case $p_{\text{divided\_cluster\_hit}} \geq \frac{c_{\text{size}}-1}{f \cdot t_{\text{window}}}$.

To compare against the fixed time window splitting, we can compute the ratio $r$ of the probabilities that the cluster was divided $r = \frac{p_{\text{divided\_cluster\_hit}}}{p_{\text{divided\_cluster}}}$. After simplification we obtain $r = \frac{c_{\text{size}}-1}{f \cdot dt_{\text{cluster}}}$. For the values of $dt_{\text{cluster}} \approx 25 \cdot 10^{-9}$ s and high hit frequencies $f \approx 10^7$ hit/s, $r$ is smaller than one only for $c_{size} < 0.25$ (which is not realistic as each cluster has at least a single pixel). Therefore, as $r > 1$ for our application $p_{\text{divided\_cluster\_hit}} > p_{\text{divided\_cluster}}$ Notably, we only provided a lower bound on $p_{\text{divided\_cluster\_hit}}$ which is reached only if none of the clusters are temporally overlapping, which is rarely true in practice. This would further increase the value $p_{\text{divided\_cluster\_hit}}$, making this approach even less effective.

We can overcome this problem by setting a closing time $t_{\text{close}}$. Instead of instantly closing the buffer after a predefined number of hits, we could mark the time of the last hit as $t_{\text{last}}$ and set the border time as $t_{\text{last}} + t_{\text{close}}$. If the $t_{\text{close}}$ is larger than the average cluster timespan, the chance of splitting the cluster is reduced. (We will likely not split the cluster open at $t_{\text{last}}$, and the chance of splitting a cluster converges to the chance of split by fixed $t_{\text{window}}$.). Another problem arises when we decide to split the hits before they are fully ordered. With the 'dynamic' time windows, we cannot correctly split the hits as we decide the window size only for the currently open window. Even for this problem, there exists a solution: for $t$-ordered data, we can precompute the window borders for the next $\left\lceil \frac{t}{t_{\text{window}}} \right\rceil$ windows and modify the window borders dynamically with a 'lag' of size $t$. Unfortunately, we expect that $t \gg t_{\text{window}}$, which makes the 'lag' of dynamic time window adaptation too long, considering the fact that the dynamic time window was meant to quickly react to a significant change in a hit frequency (This is the place where worker split imbalance is likely to happen).

## 3.3 Merging the datastreams

After the hits were split between workers and clustered, we now need to merge multiple data streams into a single one while also completing the clusters that were divided in the splitting workers because they lay at the border. It is critical that this process is sufficiently fast so it does not outweigh the speed advantage gained during parallel clustering. We propose multiple approaches to this task. The baseline approach describes the merging algorithm performed by a single worker, while the remaining merging approaches modify the baseline so that it can run in parallel. In this section, we will restrict our analysis to the data that was split temporally if not stated otherwise.

### 3.3.1 Baseline merging

The goal of the merging algorithm is to identify which of the clusters lay at the border (of the time window) and thus were split between different clustering workers. Additionally, the merging needs to complete these clusters and return all clusters in the time-sorted order.

The algorithm for merging is shown in Algorithm 2. The idea is to temporarily store the clusters, ordered by the time of arrival of the first pixel while keeping track of three additional attributes of the cluster. First, we store a flag (*border*), indicating if the cluster lies near the border. Second, we store the information about the location of the previous border cluster (*borderOffset*) relative to each cluster (the number of non-border clusters preceding each cluster). This allows us to iterate over bordering clusters quickly. And third, we store the validity flag (*valid*). This flag indicates if the cluster contains valid data or if it has already been merged with some other cluster. Instead of invalidating a cluster, it may seem natural to remove such a cluster from the container completely. However, the remove operation likely has non-constant time complexity, depending on the specific type of sorted container ($O(n)$ for arrays, $O(\log(n))$ for red-black trees ...). After we receive the cluster, there are three possible options.

- The cluster does not lie at the border, and there are no open clusters at the moment. In this case, we know we can output the cluster directly. Here, no merging is needed, and all clusters with the lover time of arrival of the first hit were already outputted (otherwise, they would be among open clusters)
- The cluster does not lie at the border, and there are some open clusters at the moment. Then, we add the clusters to open clusters. It is important to note that we can receive non-border clusters in between two border clusters that are mergeable, as illustrated by Figure 3.3. Because of this, even if the cluster is not at the border, we cannot directly output it, as it could violate the time orderliness (some open bordering cluster might precede it).
- The cluster lies at the border. In this situation, we process it by Algorithm 3.

The clusters older than $dt_{\max}$ are periodically removed from the open clusters and written to output.

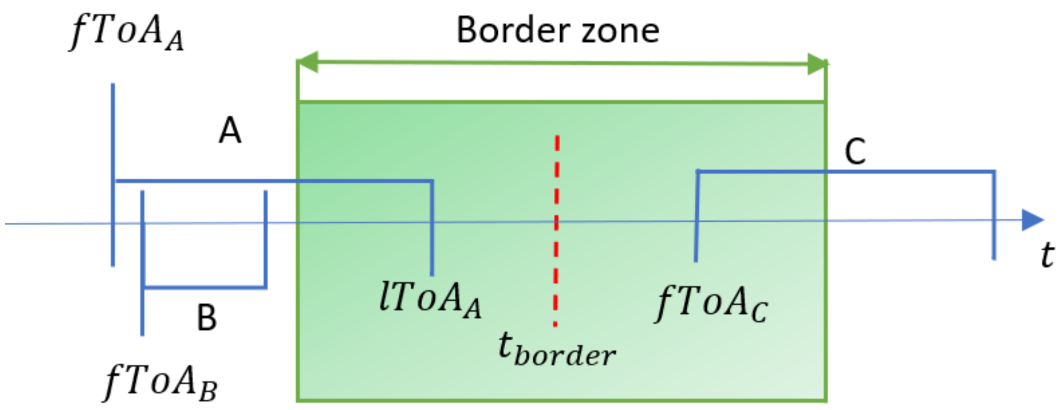


Figure 3.3: Scheme displaying three clusters $A, B$ and $C$ on a time line. The clusters $A$ and $C$ are mergeable border clusters, and the cluster $B$ is considered a non-border cluster. The labels *fToA* and *lToA* denote the ToA of the first and the last hit in the corresponding cluster. Here, the cluster obtained by merging clusters $A$ and $C$ should be outputted before the cluster $B$.

**Algorithm 2** Merging clusters

```
 1: $openClusters \leftarrow \emptyset$
 2: $borderOffset \leftarrow 1$
 3: while $dataSource$.isNonEmpty() do                ▷ iterate over clusters in dataset
 4:     $inputCluster \leftarrow dataSource$.next()
 5:     writeOldClustersToOutput($openClusters$)
 6:     if isOnBorder($inputCluster$) then                    ▷ processing border cluster
 7:         $inputCluster.borderOffset \leftarrow borderOffset$
 8:         processBorderCluster($inputCluster$)
 9:         $borderOffset \leftarrow 1$
10:     else                                           ▷ processing non-bordering cluster
11:         if $openClusters.count == 0$ then writeToOutpout($inputCluster$)
12:         else
13:             $inputCluster.borderOffset \leftarrow borderOffset$
14:             $inputCluster.border \leftarrow false$
15:             $openClusters$.add($inputCluster$)
16:             $borderOffset \leftarrow borderOffset + 1$  ▷ updating the offset to previous
    border cluster
17:         end if
18:     end if
19: end while
```

To process the border cluster, we iterate over the open border clusters and select those which can be merged with the current cluster. It is necessary to realize that multiple clusters might be mergeable in this process. (A minimal example of a cluster that would be split into three clusters is a three-pixel cluster in the shape of a horizontal line. The times of arrival of the pixels from left to right would be $t_{\text{border}} - \varepsilon, t_{\text{border}} + \varepsilon, t_{\text{border}} - \varepsilon$ for some small value of $\varepsilon$.) During the merging, we combine all the clusters into a single cluster, storing it in the open clusters at the position of the cluster with the smallest first time of arrival among the merged clusters. This way, we keep the clusters sorted in time. Then, we invalidate all other merged clusters.

**Algorithm 3** Processing border clusters

```
1: constants $dt_{\text{max}}$, $maxTimeSpan$
2: $borderOffset \leftarrow 1$
3: procedure PROCESSBORDERCLUSTER($inputCluster$)
4:     $inputCluster.borderOffset \leftarrow borderOffset$
5:     $clusterIndex \leftarrow openClusters.count - borderOffset$
6:     $currentCluster \leftarrow openClusters[clusterIndex]$
7:     $clustersToMerge \leftarrow \emptyset$
8:     while $clusterIndex \geq 0$ do
9:         if $currentCluster.valid \wedge$
10:    $inputCluster.first\_toa - currentCluster.first\_toa > dt_{max} + maxTimespan$
11:            then
12:            $break$
13:        end if
14:        if $currentCluster.valid \wedge$canMerge($inputCluster$, $currentCluster$) then
15:            $clustersToMerge$.add($currentCluster$)
16:        end if
17:        $clusterIndex \leftarrow clusterIndex - currentCluster.borderOffset$
18:        $currentCluster \leftarrow openClusters[clusterIndex]$
19:    end while
20:    if $clustersToMerge.count = 0$ then
21:        $openClusters$.add($inputCluster$)
22:    else
23:        $oldestToMerge \leftarrow clustersToMerge$.last()
24:        $clustersToMerge$.removeLast()
25:        for $clusterToMerge$ in $clustersToMerge$ do
26:            $oldestToMerge$.mergeWith($clusterToMerge$)
27:            $clusterToMerge.valid \leftarrow false$
28:        end for
29:        $oldestToMerge$.mergeWith($openCluster$)
30:    end if
31: end procedure
```

During this process, we still have not discussed how we efficiently decide if two clusters are mergeable. This is done on line 13 of Algorithm 3. Our algorithm for finding mergeable clusters focuses on quickly filtering out clusters that cannot be merged. For that, we use a cascading approach with three steps, progressively increasing in temporal complexity.

1. If the temporal distance of clusters is higher than the maximum temporal

distance $dt_{\max}$, they cannot be merged. The temporal distance $d_{c_1,c_2}$ between clusters $c_1$ and $c_2$ can be defined as $d_{c_1,c_2} = max(min(c_1.startTime - c_2.endTime, c_2.startTime - c_1.endTime), 0)$. This step can be performed in constant time.

2. If the bounding boxes of the clusters do not touch, then the clusters are not spatially neighboring, which implies they cannot be merged. Here, the bounding box is the minimal rectangle (with sides parallel to the sensor) that surrounds the cluster. The bounding box is computed only once when the cluster is added to open clusters. Thus even though the computation of the bounding box takes linear (with respect to cluster size) time $O(n)$, the query to check the intersection of the bounding boxes can be evaluated quickly in constant time.

3. If both of the above conditions fail, we perform the full merge check. A naive implementation of checking each pair of pixels could take $O(n_1 \cdot n_2)$ time, where $n_1, n_2$ is the number of pixels in the clusters. Instead, we perform the intersection check by storing the pixels from the intersection of the bounding boxes and the bigger of the two clusters in a 2D array. We then enumerate all pixels of the smaller cluster and check if any of its spatial neighbors is contained in the 2D array. The reason behind the selection of bigger/smaller clusters is that the neighbor checking is more computationally expensive than inserting the cluster into the 2D array. Without the loss of generality, suppose $n_1 > n_2$. The neighbor checking takes $O(8n_1)$ time because of 8-neighbors while inserting and deleting a cluster from the 2D array is $O(2n_2)$. Together, this process has $O(n_1 + n_2)$ time complexity.

### 3.3.2 Tree-based merging

With the increasing number of clustering workers, it is expected that, at some point, the merging worker will become overloaded. To deal with this problem, we propose a method for parallelization of the merging worker. The first approach we call tree-based merging. The tree-based merging was named as it is because the data flow resembles a binary tree. Let us illustrate this by an example. Suppose we use four clustering workers $c_0, c_1, c_2$ and $c_3$, all working in parallel. The index of the worker represents the order of assigning chunks of data split temporally. After assigning the chunk to $c_i$, we assign the following chunk to $c_{(i+1) \, mod \, 4}$. In this case, our merging will consist of two layers. In the first layer, we have two workers $m_{01}$ and $m_{23}$, each merging the output only from two of the clustering workers. However, in this layer, clusters split by the border between the second and the third worker are not merged (similarly to the clusters between the fourth and the first worker). Therefore, we introduce the second layer. The clusters near these borders are forwarded to layer two, which only has a single worker that handles the merging of these clusters. This example architecture is shown in Table 6.1.

It is not difficult to generalize the example for the case with an arbitrary number of clustering workers $n$. On the $i$-th layer, we will have $\frac{n}{2^i}$ merging workers, each merging the output from two workers from the previous layer. Given $i$-th layer, each worker handles merging blocks of time of size $t_i = 2^i \cdot t_{\text{window}}$. However, we can stop the tree merging at any layer and put there only a single worker that handles the merging of all blocks of size $t_i$ from the previous layer. Nevertheless, each layer implies additional copying of the clusters between workers. Moreover, this approach produces a single data stream at the end, which might not be required by the user. It can happen that the next steps of the analysis can be performed in parallel. In such a case, merging the data streams into a single stream and then splitting them again for data analysis seems pointless. One might argue that we could obtain multiple data streams (and preserve parallelization) by not using the last worker on the final layer, which is indeed true. However, such a data stream would contain incomplete clusters as the borders between the blocks of size $t_i$ have not been checked for clusters divided by the border.

### 3.3.3 Single-layer merging

As opposed to tree-based merging, single-layer merging is an approach that can merge all non-complete clusters utilizing only one layer. Again, the best way is to start with an example. Let us assume we use four clustering workers. For each of these workers, we create a merging worker. Suppose we name these workers $m_0$, $m_1$, $m_2$ and $m_3$. Then, each merging worker is assigned two 'neighboring' clustering workers. In this case, each worker $m_i$ processes data from clustering $c_i$ and $c_{(i+1) \bmod 4}$. Generalizing this scenario for an arbitrary number of workers $n_{\text{workers}}$ is simple. We only replace the number 4 with the value of $n_{\text{workers}}$. This implies that each clustering worker splits the data between two merging workers. The split of data is done by the time of arrival of the first hit. Namely, the first half of the time window is assigned to one merging worker and the other half to another. By splitting the time windows in half, we implicitly assume that the clusters do not span more than $t_{\text{window}}/2$. Otherwise, if we have clusters that span more than $t_{\text{window}}/2$, we might not be able to reconstruct it. One such non-reconstructible cluster could start in the first half of the window, span through the whole second half of the window, and even extend to the start of the next window. Such a cluster (as based on the arrival time of the first hit) would be assigned to a merging worker that only merges the current window with the previous one. This way, we would never find the possible hits from the next window that could be merged with this cluster. However, in practice, it often holds $t_{\text{window}}/2 > dt_{\text{cluster}}$, making the assumption behind the approach reasonable. Compared with tree-based merging, for single-layer merging, it is easier to split the data between workers evenly, allowing better utilization of computational power. The output of this merging consists of multiple streams of completed and time-ordered clusters. On the other hand, the tree-based approach can also reconstruct clusters with an arbitrarily large timespan.

## 3.4 Parallelization data flow

In this part, we will combine all discussed concepts from Chapter 3 to describe a data flow between the workers that participate in the clustering. The workers can be seen as nodes, and here we will define the edges which represent the flow of the data, together forming a directed acyclic graph (DAG). From Section 3.1, we introduce nodes like 'reader,' 'calibrator,' 'sorter', 'clusterer', and 'writer'. These workers were initially connected in a serial manner. Then we decided to parallelize by splitting the data as shown in Section 3.2. From that, we obtain multiple 'clusterer' nodes and a 'merger' node. Additionally, we noticed that if the 'sorter' node becomes overwhelmed, we can modify the architecture and perform the split even before sorting in the 'calibrator' or 'reader' node. And last, we could also parallelize the clustering as shown in Sections 3.3.2 and 3.3.3.

#### Controller node

Apart from the nodes mentioned above, we introduce an auxiliary 'controller' node, whose task is to create the data flow graph and periodically check the amount of data being processed at specific timesteps. This is usually necessary when processing loads of data simultaneously with only limited operational memory. To clarify, let us first define the throughput of a node as the number of hits it can process in a specified time window. Suppose we have two nodes – $P$ is the producer node, and $C$ is the consumer node. In the ideal case, for the throughputs $v_P$, $v_C$ of the nodes it holds $v_P \leq v_C$. In this scenario, the queues between nodes would be nearly empty and the memory consumption would be minimized. Unsurprisingly, this is not always the case, as each step of the clustering algorithm might have a different complexity. (It is possible to merge some nodes performing less expensive computations together at the cost of decreased modularity and extensibility. We would like to avoid that, as the algorithm is split into logical steps, which can be performed in parallel and possibly reimplemented, which is difficult for the 'merged' nodes). If $v_P > v_C$, the data in the queue between $P$ and $V$ will likely start accumulating. (this is also known as 'backpressure' [17]). In such a case, the controller will inform the reader (the source node) to wait until some part of the data is processed and does not need to be stored in memory. It is a safeguard that tries to prevent running out of memory during the processing at the cost of possibly losing some part of the input data (in the online case, the data in the UDP socket can become lost if it is not processed within a specific time window). We summarized all of the graph nodes in Table 3.1.

| Processing nodes | |
|---|---|
| Name of the node | Node description |
| reader (R) | Reads the input file and feeds the data to the rest of the computational graph. |
| calibrator (C) | Performs the energy computation based on the values obtained from the calibration. |
| sorter (S) | Fully sorts a $t$-ordered sequence of hits. |
| clusterer (C) | Groups the hits into clusters based on temporal and spatial similarity. |
| merger (M) | Reconstructs the clusters incorrectly divided by the data split. |
| writer (W) | Writes the produced clusters to a file in a predefined format. |

Table 3.1: Overview of the nodes used in the clustering processing graph.

### Termination of the processing

Additionally, if all the data are processed, we stop each node using the 'poison pill' pattern. This is a pattern that allows a graceful shutdown of the distributed process. Here, the source node (in our case, it is the reader) generates one last additional piece of data that carries a special flag indicating the end of the process (known as a poison pill). Each node then propagates this particular piece of data to the other nodes. In the case of a node with multiple input nodes, it only propagates the poison pill after receiving it from all of its input nodes. Otherwise, some of the input nodes could still contain unprocessed data. This way, the terminate signal propagates through the graph until every node terminates its job.

# 4. Bounding box clustering

In this chapter, we describe a different approach to clustering. It utilizes an observation about the hits from physics – If a high energy hit is measured by the detector, we can likely expect to measure more hits in the spatial neighborhood of this pixel. This is known as a 'halo effect' [18], which we try to use to our advantage. For an example of the cluster halo, see Figure 4.1. Nevertheless, this approach is based on an important adjustment to the original clustering task. We no longer require the clustering to produce exact clusters as long as a significant portion of the hits is clustered correctly. In other words, we are implementing an approximate clustering, trading quality for performance.

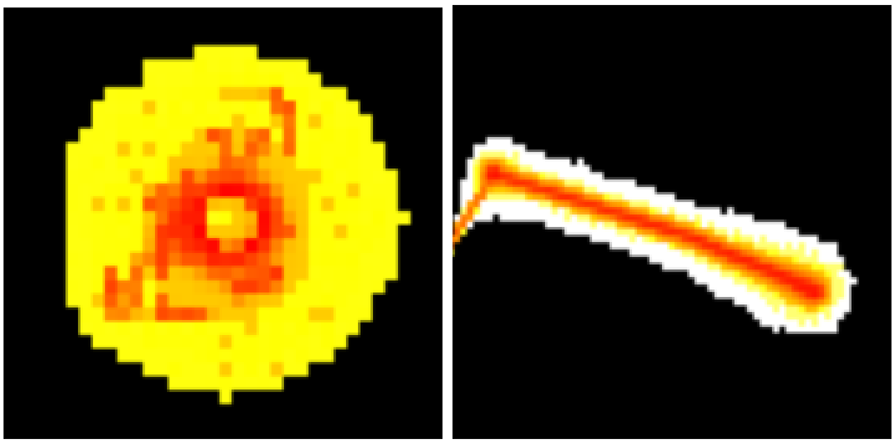


Figure 4.1: Images of the clusters with a halo. The halo pixels are marked by a yellow and white color, indicating a low deposited energy.

### Splitting data into chunks

The first difference between bounding box clustering and standard clustering is the fact that bounding box clustering can only process data divided into chunks, as opposed to standard clustering, which does not require it. If online processing of a potentially unbounded stream of hits is desired, the stream needs to be first cut into multiple buffers. Notably, such division of data we already discussed in the chapter about parallelization in Section 3.2.2 and Section 3.2.1.

### Tile clustering

Another concept we utilize in the bounding box clustering is the relaxation of the spatial connectivity of pixels in a cluster to the spatial $d$-connectivity of pixels. We say a cluster is $d$-connected if for each pair of hits $(h_a, h_b)$ in a cluster there exists a path of hits $(h_1, h_2, ...h_n)$ such that $h_1 = h_a$, $h_n = h_b$, and the spatial distance between $h_{i+1}$ and $h_i$ for every $i \in \{1, 2, ..., n-1\}$ is at most $d$ pixels. The spatial distance can be measured using Manhattan or Euclidean metrics.

This $d$-connectivity can be useful mostly in the following three cases.

- We are clustering only a subset of pixels in a cluster. Because of that, the spatial 1-connectivity might be broken, even though hits in the subset might still be $d$-connected for some small value of $d$.
- There are some insensitive pixels in the sensor that we want to ignore. (These pixels are also called 'dead' in the pixel detectors, for instance, see [19])
- During the incremental building of a cluster, we want to avoid unnecessary merging of clusters. In standard clustering, usually, before the full cluster is formed, a couple of steps of merging the sub-clusters are required. It is because the hits can arrive in arbitrary order. This means that even if the arrived hit is only two pixels away from a huge cluster, we do not add it to the big cluster, although there is a high probability that the hit connecting these two clusters will eventually arrive. Then, these clusters need to be merged, which could be avoided by using tile clustering.

The idea of tile clustering is a small modification of standard clustering. First, we divide the area of the sensor into multiple tiles of size $d \times d$ pixels. And rather than storing the list of clusters for each pixel, we store it for each tile. Then, when performing the neighbor check to find neighboring pixels, we look for the neighboring tiles instead. The rest of the algorithm stays as it is in the standard clustering. Naturally, relaxing on spatial connectivity can produce invalid clusters if two clusters are very close to each other (that is the reason why it is only an approximate clustering method).

### The algorithm of bounding box clustering

With tile clustering defined, it is time to describe the bounding box clustering. This algorithm has the following steps:

1. Obtain a buffer of hits from the input.
2. Select hits with energy above the specified threshold $e_{\mathrm{t}}$. We expect these high-energy hits to create a halo effect. These high-energy hits often form the backbone of the cluster.[Image of a cluster with halo]
3. Perform tiled clustering on the filtered hits with the tile of size $d \times d$. Because the selected hits might not be spatially connected, we use the relaxation to $d$-connectivity.

4. Draw a bounding box around each cluster. Additionally, we add a 'padding' of size $p$ to the bounding box by extending it to each side. An interesting modification would also be to allow tilted bounding boxes which are not parallel with the axes of the sensor.
5. Iterate over remaining hits and add them to a cluster if they lie in its bounding box.
6. If there are any non-clustered hits, lower the value of $e_t$ (and possibly $d$) and return to step 2. Otherwise, return the clusters.

We can see that we still need to perform standard clustering on some fraction of hits. Despite that, all of the hits located in the bounding box do not need to be clustered in a standard manner as we add them directly to the cluster. The reason why this approach could be faster is skipping the neighbor search, which occurs in standard clustering for each hit. On the other hand, keeping track of the bounding boxes might require additional computational power, so it is difficult to predict if this method actually speeds up the clustering. Important parameters that affect the quality and the speed of the clustering are the threshold value $e_{\mathrm{t}}$ and the tile size $d$. We do not provide any clever method of setting the value of these parameters. Nevertheless, we can estimate the value of the parameters experimentally.

# 5. Clustering triggers

The purpose of this chapter is to introduce another method of acceleration of the clustering. Similarly to Chapter 4, we will relax the clustering task. However, this time we do not relax on the quality of the created clusters. Instead, we will selectively decide which data should be clustered and ignore the uninteresting parts of the data stream. The rule which describes when to initiate clustering we call the trigger.

We will start by introducing the motivation behind the triggers and some properties that we require from the triggers. Then, in Section 5.1, we discuss a very simple example of a hit-based trigger. In the following Section 5.2, we will generalize this example to a more useful window trigger. In some cases, it may be useful to measure the changes in the radiation field instead of the absolute values of features. This is discussed in Section 5.2.1. And last, we discuss two types of triggers – explicit and implicit. The explicit triggers directly utilize the user's knowledge about the properties of the stream of hits, as shown in Section 5.2.3. The other group of triggers infers the desired properties implicitly after the user provides examples of interesting series of hits. For such an inductive approach, it seems natural to employ methods of machine learning.

Let us first discuss the motivation behind the triggers for clustering. The concept of a trigger is already frequently used in the area of high-energy physics [20]. Our triggers could be used in various scenarios. An example is measuring a beam of particles, which is turned on only during some time intervals. Additionally, there is natural background radiation. Hence, we are only interested in the data from the beam, and therefore clustering of the background is not required. A simple trigger could estimate if the beam is on and start the clustering. Another example can be a measurement of radiation, where we want to detect various types of particle showers. Here, the physicists might consider some of the particle types more interesting than others. Triggers would allow them to filter and only cluster some specific types of particle showers. And a last example could be the case where we do not apriori know what kind of radiation field we consider interesting, but we try to detect significant radiation field changes. Here, the triggers could detect the change and only initiate clustering in that case. We try to address all of the mentioned examples with trigger clustering.

When creating a clustering trigger, we need to keep the following properties in mind:

1. Efficiency. If the evaluation of the trigger were more computationally expensive than the clustering of the hits, then the usefulness of the trigger

would be disputable.

2. Customizability. As there are different possible applications of the triggers, we require them to be adjustable to the needs of the user. Each user should decide (not necessarily explicitly) what piece of data is worth clustering.

It is important to note that apart from possibly reducing the computational complexity, by triggering clustering, we can also save the storage space required to store the clustered data. Thus another advantage of this selective clustering is data reduction.

## 5.1 Energy-hit-based trigger

One of the key properties of hits is energy. A simple trigger could monitor the energy of hits in the data stream and only start clustering when a specified threshold is reached. In the ideal case, instead of high-energy hits, the physicists would like to cluster the hits that form high-energy clusters. Obviously, we do not know which hits are part of such clusters until we perform the clustering. For example, there can be clusters where none of the hits reach the energy threshold, but the number of hits in this cluster is so high that this cluster would be considered a highly ionizing cluster. With detecting only the energy of hits, we would not process such clusters, which is a limitation of this approach. However, it is very frequent that highly ionizing clusters also include high-energy hits.

A naive way to implement this algorithm would be monitoring the energy of hits and discarding the hits until we receive a hit with energy above the threshold. Nevertheless, these high-energy hits may only arrive after some other low-energy hits from the same cluster, which we already discarded. Because of that, we need to keep some portion of hits in a buffer for a period of time until we are sure no new cluster could possibly contain them. After starting the trigger, we now need to discuss when the clustering should be stopped. The most straightforward way is to set a time $t_{\text{trigger}}$, after which the clustering stops. If the trigger activates again when it is already on, we simply extend the duration of the trigger. In this simple energy trigger, there is also another option when to stop clustering. It is possible to keep track of the number of currently processed clusters containing high-energy pixels. And when all of these clusters are processed, we can stop the clustering.

## 5.2 Generalized window trigger

In this section, we will try to generalize the energy-hit-based trigger. Unfortunately, the hit itself provides us with only very limited information. Apart from energy, it usually contains only its spatial and temporal coordinates. With these pieces of information, detecting complex changes in the radiation field is difficult, if not impossible. Because of that, we decide to collect multiple hits before making the decision to trigger. More specifically, we monitor the data stream for a specified time $t_{\text{window}}$, and apart from the properties of the hits, we also monitor the statistics of these properties. This gives us a better chance to distinguish various types of radiation. When computing statistics, we need to keep in mind efficiency because of the limited time in which the computation must finish. To create a lower-dimensional representation of each window, we compute the following:

- Mean and standard deviation of the spatial coordinates. These properties aim to detect changes in the particle beam direction. With the mean, we need to be careful, as for a time window with no hits, the mean is not well defined. For that, we choose the commonly used invalid value called 'Not a Number' or NaN. The same situation can also happen for the standard deviation. Additionally, from the point of view of computational complexity, we can compute all of these features with constant memory. We only store the sums of the spatial coordinates and the sums of squares of these coordinates. Then we estimate the standard deviation (of the spatial coordinate $x$) as $\sigma_x(t) = \sqrt{\frac{\sum_{x \in w_t} x^2}{n_{\text{hits}}} - \left(\frac{\sum_{x \in w_t} x}{n_{\text{hits}}}\right)^2}$ with $w_t$ being the window at time $t$.

- The number of hits. By keeping track of the number of hits in a fixed time window, we could detect the changes in the particle frequency.

- The maximal energy, the mean energy of the hit, and the energy distribution. It was shown that even simple per-pixel metrics like the energy left in the pixels have proven as a solid measure for the separation of electrons and protons [21]. Moreover, we need to choose the binning of the energy hits carefully. By binning, we mean the process of discretization of continuous variables – in our case, the energy of the hit. The number of bins needs to be small enough so that the processing of each window is still efficient. On the other hand, we would like to make the bins narrow enough so they reflect the possible changes in the data stream as best as possible. As the energy of the window is theoretically unbounded, the fixed size of the bin will not suffice. Additionally, from the point of view of the experimental physicist, we can afford to lose precision in binning with the increasing energy of the hit. To fulfill all of the binning requirements, we choose logarithmic binning. This gives us higher precision in the lower-end energy spectrum while also saving space and processing time for the (often sparse) hits in the high end of the spectrum.

- The average size of the 'temporal' cluster. The set of hits $\{h_1, h_2, \dots h_n\}$ (without the loss of generality $\text{timeOfArrival}(h_i) \leq \text{timeOfArrival}(h_{i+1})$) form a temporal cluster if $\text{timeOfArrival}(h_{i+1}) - \text{timeOfArrival}(h_i) < dt_{\max}$ where $i \in \{1, 2, \dots n-1\}$ and $dt_{\max}$ is a constant that depends on the properties of the detector. In contrast to the standard cluster, we resigned on the spatial neighborhood condition. In contrast to standard clustering, the computation of the temporal cluster is very fast. For that, we do not require any auxiliary data structures. First, we sort the hits in time, and then we close the cluster if no hits were detected for more than $dt_{\max}$ units of time. For low-frequency data sources, it provides a simple approximation of the true clusters. With increasing the frequency of the source, however, we expect an increase in incorrectly merged clusters. For such radiation sources, this method provides a lower bound on the number of clusters in a window (and thus an upper bound of the true average cluster size, as each temporal cluster can be mapped to one or more standard clusters).

### 5.2.1 Differentiating the features

After we split the data into time windows and compute the statistics, we need to use the statistics as window features to decide if the data should be clustered. In the case where it is desired only to cluster some specific types of fields, we could use the values of the statistics directly. Nevertheless, in some situations, when aiming to trigger if the radiation field changes, the statistics themselves are not sufficient. Instead, we need to compute how they change over time, also known as the differentiation of the features with respect to time. Let us now define the feature differentiation as the mapping $dt : t \to f_t - f_{t-1}$ with $f_t$ being the feature $f$ of a window with index $t$.

The feature differentiation, as we defined it, has an unwelcome property. It is very sensitive to noise in the data. More specifically, during the measurements, we can encounter many different sources that add noise to our data, including background radiation or scattering of the particles into secondary particles. Additionally, there can be imperfections in the sensor that introduce additional noise factors. By observing the differentiation of the features in such a noisy environment, we can expect to measure high difference values most of the time of the measurement. To counter the effect of the noise, we propose to differentiate with respect to the multiple $k$ previous time windows. So instead of considering only $f_{t-1}$ we consider $\{f_{t-1}, f_{t-2}, \ldots f_{t-k}\}$. Now, we need to decide how to aggregate this set of time windows. We suggest finding the median of these windows to differentiate with respect to their value. Another option would be to compute the mean, but we consider the median more resistant to noise [22]. Together, we can define the temporal window differentiation as $dtw : t \to f_t - med\{f_{t-1}, f_{t-2}, \ldots f_{t-k}\}$.

## 5.2.2 Explicit interval trigger

Let us now discuss the creation of the trigger from the measured data. The first and simplest option is to let the user decide what values of the statistics should start the trigger. We can call this trigger an explicit trigger because we let the user explicitly choose which window statistics values are considered worth clustering. For each feature, the user can choose an interval of interest. If there is a window where all of its features belong to the user-specified intervals, we start the trigger. This, however, restricts the user only to select a single interval for each feature. Luckily, we can extend this option in a simple manner – the user could choose multiple intervals that would be linked by the conjunction and disjunction logic operators. In general, the user creates the trigger formula in disjunctive normal form (DNF), namely $\bigvee_{c \in clauses} \bigwedge_{f \in features}$: $w_f \in I_{f,c}$ where $w_f$ is the value of the feature $f$ for the time window $w$ and $I_{f,c}$ is the $c$-th interval for the feature $f$. The ability to combine multiple clauses enables the user to select arbitrary regions of the feature space as features worth clustering. As a matter of convenience, not all of the intervals $I_{f,c}$ must be specified by the user directly. If the interval $I_{f,c}$ is not specified, it is automatically considered to include all values, thus setting $I_{f,c} = (-\infty, \infty)$.

## 5.2.3 Implicit ML-based trigger

Even though the approach discussed in Section 5.2.2 might seem like a very flexible trigger, there are scenarios where it is not very practical. For instance, the user might not be able to specify the intervals of interest explicitly but rather present some examples of windows that should be clustered. Thus, we are given the windows of interest that define the triggering condition implicitly (likely not in a unique way). Our task is to infer this triggering condition. Because this trigger is not created explicitly by a user, we decide not to restrict the triggering condition to the form of a union of hypercuboid regions of the feature space. Instead, as we aim to learn from examples, machine learning (ML) seems to be a natural choice.

With machine learning, we might want to revisit the features we used for trigger clustering. As the values of the features of interest are no longer directly specified by the user, we could try adding new different features. One of such features of a window could be a 2D matrix where each value at index $i, j$ corresponds to a total energy of the pixel received in the particular time window. It contains both spatial information and information about the shape of the clusters. Such data would be easily processed as an image by machine learning models that are intended for image classification. Additionally, the creation of such a matrix would be relatively inexpensive. Initialization of the matrix can be done only once at the start of the program, and during runtime, we would only modify the values of hit pixels. However, the problem arises when we try to process such a matrix by ML methods in a short time window. Despite the fact that there is a chance that this image processing idea is intractable for our application, we would like not to abandon it completely. One of the options could be to downscale the resolution of such an image to save processing time. The decision if this is a viable option might depend on the particular ML model, and it is best to explore it experimentally.

Before discussing the ML models, we should first discuss the creation of the training data. For that, we prepared a simple application where the user can create the training data without much effort. The data creation consists of multiple steps:

1. **Loading the data.** The user selects a data file that contains the data relevant to the trigger. Standardly, we expect the user to provide the data in raw text file format together with a calibration file used for the computation of the deposited energy of the hits. Notably, in cases where the data is split into multiple input files, it is possible to process them one by one and create a single trigger training dataset. In this step, the user can also set a few parameters like the size of the window. Furthermore, the user can choose to only compute the values of the features or to calculate feature differentiation instead.

2. **Computation of the window features.** In this step, the input data is processed and the features are computed based on the arguments specified in the previous step. All of the computed features are stored temporarily and displayed to the user in a table.

3. **Exploration of the features.** Here, the user should observe the values of the statistics and displayed plots to asses which windows of data are intended to be clustered.

4. **Selection of the windows.** The user has two options for the selection of windows. First, the user can either select the windows manually by clicking on the windows in the table. Even though this process might seem tedious, in some cases there might not be a better approach.

   The second option is to select windows by specifying the intervals for the window features. Only the windows with features inside the selection intervals are added to the selection. Even though this process resembles the explicit interval trigger, it is not equivalent. In the explicit interval trigger, the trigger is activated only in the hypercuboid (potentially unbounded) regions of the feature space. In the ML trigger, we allow the model to choose the trigger regions based on the data points. In general, the user is expected to combine the manual selection and interval selection to find the windows of interest efficiently.

5. **Saving the window features.** After the desired windows are selected, the user selects a class label and saves them to a file. Here, the class label is a name for the selected windows. Later, just before the training of the model, it is possible to choose labels that contain the interesting windows and the ones which do not. It is also important to include at least a single class that contains negative or 'uninteresting' data. These can be selected the same way as the windows of interest. Consequently, for supervised triggers, it is needed to create at least two classes – 'trigger' and 'no_trigger'. For unsupervised triggers, it is sufficient to create a single class of data.

Regarding the choice of ML methods, we propose multiple alternatives:

- **Multi-layer perceptions (MLP).**[23] [24] The MLP is a neural network that consists, as the name suggests, of multiple layers of neurons, each connected with neighboring layers. Each neuron receives the activation of the neurons from the previous layer, multiplied by the corresponding weight vector. The input is processed by an activation function (commonly used activation functions are Rectified linear unit (ReLU), Exponential linear unit (eLU), the sigmoid function, and more, see [25]). In this step, the information in the network propagates in a single direction, which is known as the inference phase. Then, in the learning phase, the gradient of the weights with respect to the output error is computed and propagated backward through the network. The gradient is then used to adapt the weights in the network with the goal of minimizing the error of the network output. There are also other kinds of neural networks available, but the simplicity of the perceptron makes it a good candidate for fast inference time.

- **Support vector machines (SVM).**[17] [26] The SVM tries to find separating hyperplanes between the data points that minmize the classification error and maximize the margin between these classes. Such task can be formulated in the form of constrained optimization enabling the use of Lagrange multiplier method [27]. It also is capable of handling linearly non-separable classes by utilizing the so-called kernel trick – mapping the data points into multidimensional feature space with a non-linear mapping. Frequently used kernels include the Radial basis function (RBF), polynomial, and sigmoid kernels. Similarly to MLP, the inference speed of this method is expected to be high.
- **One class support vector machines.**[28] In contrast to the standard support vector machines, one-class SVMs are very successful at tasks like unsupervised learning and, more specifically, novelty detection [29]. In the novelty detection task, during the training, we present the model with only a single class of data. It then tries to find the hyperplane that encloses the training examples (minimizing the error) while also maximizing the distance from the origin (maximizing the margin). This leads to a constrained optimization task, similar to standard SVM. Then, during the inference phase, the task is to detect all samples that are considered as a novelty (outlier) given the training dataset. However, one-class SVMs are not inherently an online model. A nice extension would be to retrain the model periodically with new data so we adapt the definition of the novelty based on the historical data. Here, it would be important not to retrain the model very frequently as this could slow down the whole application.

### Training

Regarding the supervised training of the models, the dataset is randomly split into three parts: training, test, and validation dataset. The relative sizes of these parts are supplied by the user among the other hyperparameters. The default split is 80% for training, 10% for testing, and 10% for validation. Now let us discuss the splitting method. Suppose we have no assumption about the sizes (number of examples) of the classes selected by the user. These classes can potentially be imbalanced. To deal with the potential imbalance, we perform the split for each class separately. This way, each part of the dataset (train, test, and validation) is guaranteed to contain the same distribution of classes (not necessarily uniform). Additionally, oversampling can be used to combat the imbalance [30] [31].

#### Metrics

Concerning the metric to assess the quality of our models, we use accuracy, precision, and recall [32]. The reason why accuracy as a metric might not be sufficient is, again, the class imbalance. For datasets with imbalanced classes, the model might learn to always predict the majority class and thus yield high accuracy. This scenario is identifiable by the precision and the recall metric. Their importance might vary depending on the application. For instance, if it is critical not to miss any important clusters, then the recall of the model is important. On the other hand, if it is more important not to incorrectly trigger clustering on the uninteresting data and minimize the computational costs, then apart from accuracy, precision is the target metric. In cases where neither precision nor recall is the main target, the F1 score is a frequently used metric. After every epoch (for MLP), the user is shown the value of the loss on the training and validation dataset, together with mentioned metrics. At the end of the training, the model is tested once again on a test set to evaluate the performance of the trained model.

#### Extensibility

The best model type might vary based on the application, so we let the user choose the appropriate method. From a long-term point of view, we should be able to use more types of ML models as a trigger. To make our approach extendable, it is required to generalize over the ML models. Fortunately, there exist formats that support the serialization and deserialization of various types of models [33].

### 5.2.4 Combining triggers with parallelization

In this subsection, we discuss how to implement triggers in the context of the dataflow graph described in Section 3.4. The triggering process is inherently connected to the clustering. Our first option is to create a specific graph node named 'Trigger clusterer' that would handle trigger clustering, be it an explicit or implicit trigger. At first sight, this might look reasonable. The problem becomes evident when we try to perform the data splitting into blocks of the size defined by $t_{\text{window}}$. It stems from asynchronous clustering. If the clustering is triggered, it should persist in the triggered state for a time $t_{\text{trigger}}$. Typically, $t_{\text{trigger}} \gg t_{\text{window}}$, which means that after triggering, some hits outside of the current window should be clustered as well. Unfortunately, these hits are processed by different clustering workers. One might think that can be solved by passing this trigger information to the other workers. Nevertheless, these workers work asynchronously, which means that once a worker receives the message to start the trigger, it might already be too late. It is possible that at the moment of receiving the message, the worker has already processed the hits that should have been affected by the trigger. This issue could be handled by synchronization of the clustering nodes, which would likely negatively impact the performance. To avoid synchronization, we can modify our approach. We can create a separate worker that would act as a trigger. Evidently, this node would need to be included in the computation before the data-based split. Such a 'trigger node' would handle triggering, discarding hits while not triggered, and feeding the hits to the rest of the computational graph if triggered. Inauspiciously, this approach clearly has a disadvantage because every node on the computational path of a hit means additional copying of the data. The impact of this effect is examined by benchmarks in Section 6.2.3.

# 6. Experiments

In this chapter, we will evaluate our methods for clustering. We start with the description of the test data and models in Section 6.1. The first important criterion we will measure is the correctness of our approach – a comparison of the output of our implementation compared to the current baseline algorithm. To read more about this comparison, see Section 6.2. And then, in Section 6.3, we estimate the efficiency of the proposed clustering methods. In both parts, we used multiple datasets, as we expect both the algorithms' correctness and performance to vary based on the characteristics of the input data.

## 6.1 Experiments setup

### 6.1.1 Testing data

In order to perform the experiments, it is required to select the data which would be used as an input to our models. Our goal is to test the algorithms using a variety of cluster types. There are two factors which we can test that affect the shape of the cluster:

- **Particle type.** The type of the particle determines its charge and its structure. That defines how the particle behaves in the electric field during the measurement.

- **Particle energy** The energy of the particle depends primarily on its momentum, which is set by the radiation source.

- **Traversing angles.** By the traversing angles, we mean two angles zenith $\theta$ and azimuth $\phi$ required for a unique definition of line orientation in a three-dimensional space. For a single particle, it is difficult to accurately set the angle during measurement, but it is possible to do that on a statistical basis. Assuming the beam with the given distribution of particle directions, we can modify the shape and the mean of the distribution by simply rotating the chip while keeping the beam orientation fixed (or vice versa).

To obtain the most realistic estimate of the performance of our models, we decided to test datasets that vary in all three factors – particle type, energy, and angles.

Beside the factors influencing the cluster shape, we also decided to test different particle fluxes (the rate at which the particles traverse a fixed area). This factor can influence the effectivity of parallelization as high particle flux puts more pressure on the nodes in the computational graph.

The testing datasets included the pion, neutron, lead, and other fields, each with different energies and multiple angles. The test data were obtained during the acquisition at CERN beam lines.

### 6.1.2 Tested models

In this section, we provide a quick overview of the models we tested and references to the parts of the thesis where we discussed them. The models are summarized in Table 6.1. First, we implemented a step-based parallelization model, and then we extended it with data-based parallelization. Various merging techniques and triggers were tested. Notably, the nodes used in the computational graphs can be viewed as variables. For instance, the clusterer node can be implemented either as an exact or approximative clustering method. Typically, the outputter node would write the data onto a disk. However, during the benchmarks, it can simply receive the data and discard it to remove the bias of the I/O operations.

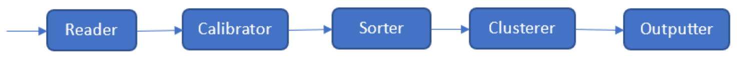
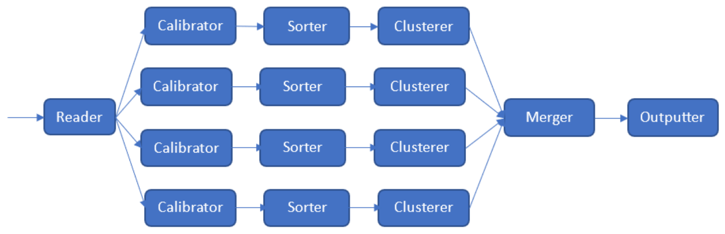
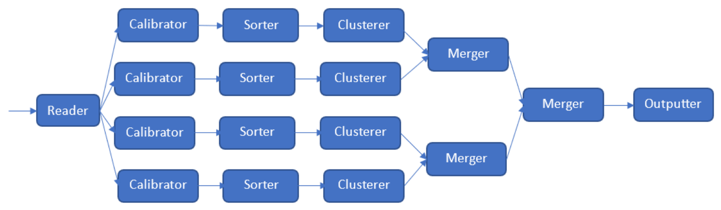
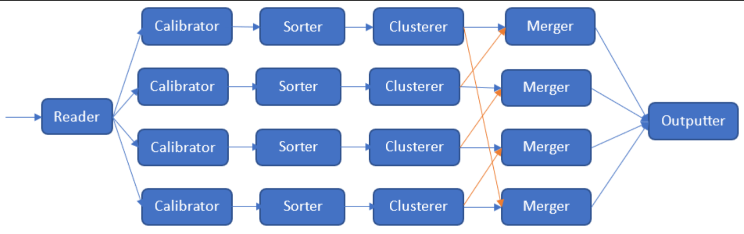
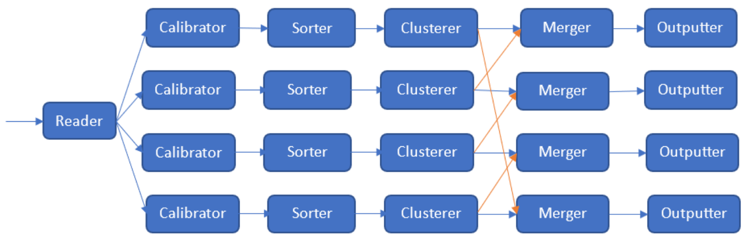
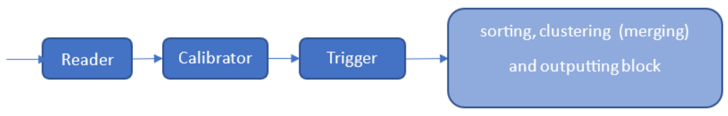
| Tested computational graphs | |
|---|---|
| Name of the architecture | The computational graph |
| Simple clusterer, Section 3.1 |  |
| Parallel clusterer with simple merging [1], Section 3.3.1 |  Reader<br>Calibrator<br>Sorter<br>Clusterer<br>Merger<br>Outputter |
| Parallel clusterer with tree merging, Section 3.3.2 |  Reader<br>Calibrator<br>Sorter<br>Clusterer<br>Merger<br>Outputter |
| Parallel clusterer with single layer merging, Section 3.3.3 |  Reader<br>Calibrator<br>Sorter<br>Clusterer<br>Merger<br>Outputter |
| Parallel clusterer with single layer merging and multioutput |  Reader<br>Calibrator<br>Sorter<br>Clusterer<br>Merger<br>Outputter |
| Trigger clusterer, Section 5.2 |  Reader<br>Calibrator<br>Trigger<br>sorting, clustering (merging) and outputting block |

Table 6.1: Overview of the tested computational graphs. In the last scheme, the block is replaced by one of the (possibly parallel) models mentioned above.

[1]If not stated otherwise, the reader is set as a splitting node, but the splitting can also be done later in the process – in the calibration or the sorting node.

## 6.2 Correctness

### 6.2.1 Metric

To verify that our implementation works as intended, we should compare it to the existing algorithms. For the comparison, it is required to define a similarity metric evaluating how well two cluster datasets match.

The first option could be to simply compare the number of clusters in both datasets. This is surprisingly effective during the development process, as it provides a quick and simple estimation of the validity of the approach. However, it is a positively biased estimator of the overlap between the two datasets (as some of the error clusters will likely cancel out in the difference in cluster count).

Then, if our goal is to see how similar the two datasets are, we can compute the intersection of these datasets. To compute the intersection, we need to define equality on clusters. We say two clusters $C_1$ and $C_2$ are equal for given $\varepsilon > 0$ if:

(i) $C_1$ and $C_2$ contain the same number of hits, and

(ii) for each hit $h_1$ in the cluster $C_1$ there exists a hit $h_2$ in $C_2$, such that for each property of the hit $p : H \to \mathbb{R}$ it holds $|p(h_1) - p(h_2)| < \varepsilon$, with $H$ denoting the set of all possible hits.

The reason for using $\varepsilon$ is to avoid mismatches between clusters only caused by the precision of floating point numbers.

While the intersection might provide good insight into how well two datasets match, to make it a useful metric, we need to normalize it so it does not depend on the size of a particular dataset. For that, we can compute the union of the two datasets. Together we form the metric named 'Intersection over union' ($IoU$), also known as the Jaccard index [34]. Given sets $A$ and $B$; $IoU(A, B) = \frac{|A \cap B|}{|A \cup B|}$.

### 6.2.2 Similarity metric computation

For the computation of the metrics, we can reuse the concept of computational graphs used for clustering. All we need is to implement new nodes and build the graph accordingly. We will only require two types of nodes. The first node, 'Cluster reader' reads the clustered files, and the 'Overlap computer' computes the similarity metric between the datasets. The computational graph consists of only three nodes and is illustrated in Figure 6.1

In general, the computation of the $IoU(A, B)$ can be done in multiple ways.

- It is possible to store one of the sets, be it $A$, as a hashed set and then query the hash set for each element of the set $B$. This operation has constant time

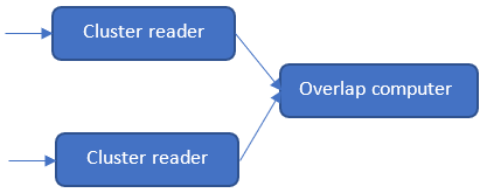


Figure 6.1: Computational graph for the similarity metric.

complexity, leading to comparison time in $O(|A| + |B|)$. A disadvantage of this approach is that the whole $A$ has to fit into operational memory, which might not always be the case.

- Another approach is to sort both sets and then read elements in ascending order. The same elements should be located close to each other in the sequence. In theory, this approach does not require the whole set to fit into memory (see external sorting [35]), but the time complexity of the sorting is $O((|A| + |B|) \cdot \log(|A| + |B|))$, which is asymptotically worse than the first approach.

For our application, however, we can utilize the fact that the cluster files are already temporally sorted. Thus we can perform the comparison computation in linear time $O(|A| + |B|)$ and constant memory[1] as shown in Algorithm 4. The function 'readNewCluster' returns the next cluster in sorted order and index of the dataset where the cluster originated. The clusters with the same value of the first time of arrival are temporarily stored in the containers $openClusters_1$ and $openClusters_2$ and periodically compared in 'compareClustersAndDispatch' method, which computes the number of matching and non-matching clusters in the sets. Additionally, it removes the matched and unmatched clusters from open clusters. In the case of clusters with distinct first times of arrival, both sets will only consist of two clusters. (A second cluster with distinct time guarantees there are no clusters with duplicate first time of arrival.)

[1] In case of clusters with the same value of the first time of arrival, we need to store all of them into memory for comparison, as they can be ordered arbitrarily. This also slows down the computation, but we rely on that occurring rarely.

**Algorithm 4** Find similarity of cluster datasets

```
1: $\varepsilon \leftarrow$ a small positive constant
2: $openClusters_1 \leftarrow \emptyset$
3: $openClusters_2 \leftarrow \emptyset$
4: while $dataset_1$.isNonEmpty() or $dataset_2$.isNonEmpty() do
5:     $(cluster, datasetIndex) \leftarrow$ readNewCluster($openClusters_1$, $openClusters_2$)
6:     if $datasetIndex = 1$ then
7:         $openClusters_1 \leftarrow openClusters_1 \cup cluster$
8:     else
9:         $openClusters_2 \leftarrow openClusters_2 \cup cluster$
10:    end if
11:    while $|openClusters_1| + |openClusters_2| \geq 2 \wedge$
12:     $|(openClusters_1 \cup openClusters_2)$.first().fToA() $-$
13:    $(openClusters_1 \cup openClusters_2)$.last().fToA()$| > \varepsilon$ do
14:                                                    ▷ fToA is first time of arrival
15:        compareClustersAndDispatch($openClusters_1$, $openClusters_2$)
16:    end while
17: end while
```

### 6.2.3 Results

For the purpose of validation of clustering methods, the models mentioned in Table 6.1 were extended by a single ‘Writer’ node which writes the cluster to a disk. The mean $IoU$ values for each method are listed in Table 6.2.

| Parallel clustering validation | | |
|---|---|---|
| Name of the architecture | The average $IoU$ score when compared to quadtree method discussed in 1 | Standard deviation of $IoU$ when compared to quadtree method |
| Simple clusterer | 0.99986 | 0.0001 |
| Parallel clusterer with simple merging | 0.99983 | 0.0002 |
| Parallel clusterer with tree merging | 0.99983 | 0.0002 |
| Parallel clusterer with single layer merging | 0.99983 | 0.0002 |
| Simple clusterer with halo-based clustering[1] | 0.854 | 0.025 |

Table 6.2: Results of the clustering validation experiment for the pion dataset acquired during measurement at CERN beam lines.

From the results, we can observe negligible differences between our implementation of parallelized clustering and the existing approach. We suppose the error primarily stems from the minor difference in the cluster definition (fixed time window vs. moving time window). Additionally, the error can be caused by the possible differences in rounding of floating point precision numbers. In the case of halo-based clustering, we see a significant number of incorrect clusters, which makes the usefulness of the approach disputable.

After validation of our clustering against existing methods, we decided to perform another validation experiment. In one of the methods for clustering, we resigned on the spatial neighborhood condition and named this approach 'temporal clustering'. Assuming the timespan of most clusters is small and the particle flux not so high, this method could work reasonably while being significantly simpler (and possibly faster). The validity of the temporal clustering for different datasets is shown in Figure 6.2. Here it is evident that with increasing hit frequency of the source, the importance of the spatial information grows as clusters start becoming closer in the temporal dimension. It depends on the temporal distances between the clusters. However, in general, as a standalone approach, purely temporal clustering does not seem to be sufficient. Notably, it could be used as a preprocessing step before the actual 'spatial' clustering.

On a side note – the spatial connectivity requirement for a cluster might not be critical for all applications. For instance, just before a particle decays when hitting the detector, it can generate multiple discontinuous tracks connected only in the temporal dimension, clustered together by temporal clustering. Even though the temporal clustering does not match the standard clustering with higher hit rates, it still possesses this usable feature.

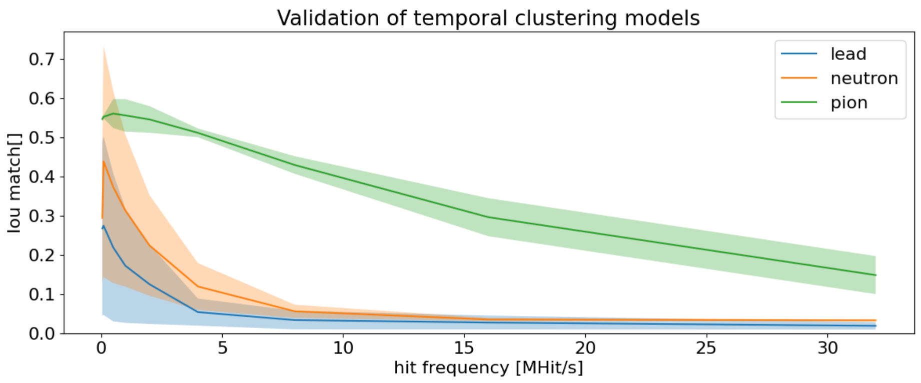


Figure 6.2: Dependence of the validity of the temporal clustering on the frequency of the received hits.

[1]Every computational graph from Table 6.1 allows replacing the clusterer node with a different clustering method. Instead of standard clustering, halo-based clustering or other methods can be used.

## 6.3 Performance

### 6.3.1 Hardware

In this section, we discuss the results regarding the speed of the clustering methods. We expect our models to run mostly on two types of devices – on a laptop with a relatively small number of cores and on a server with more computational cores. That is the motivation behind running the benchmarks twice, once for a laptop and the second time on a server. The specification of the hardware used for testing is listed below:

Laptop:

| | |
|---|---|
| Os: | Linnux Mint |
| CPU: | Intel Core i7-11370H |
| | 3.3GHz(Turbo up to 4.8GHz), 64 bits, 4 Cores |
| | L1d cache 192KiB, L1i cache 128KiB, L2 cache 5MiB, L3 cache 48MiB |
| RAM: | 8GB |
| DISK: | NVMe M.2 SSD 512GB |

Server:

| | |
|---|---|
| OS: | Ubuntu |
| CPU: | AMD Epyc-Rome |
| | 2GHz, 64 bits, 16 Cores |
| | L1d cache 1MiB, L1i cache 1MiB, L2 cache 8MiB, L3 cache 256MiB |
| RAM: | 16GB |
| DISK: | SATA SSD 512GB |

### 6.3.2 Simulation of online environment

During our benchmarks, we want to verify if the models are suitable for real-time clustering. Benchmarking the models on 'online data sources' is not feasible, as properties like hit rate are difficult to control properly. Notably, our offline clustering models can be applied to real-time clustering – it is only required to reimplement a single node, the 'reader'.

Therefore, we would like to simulate the online scenario to get a reasonable estimate of 'online' performance. With the performance, we mean the maximal speed of clustering in hits per second. To do that, we decided to exclude the cost of I/O operations from most of the benchmarks. The cost of output operations can be neglected by not including the 'Writer' nodes in the computation graph. The cost of input (reader) operations can be reduced by utilizing the 'recurring reader', which loads the data before the start of benchmarks and then generates hits repeatedly, offset by the fixed time without any additional reads from the disk. The time offset can then be set as a time difference between the first and the last hit stored in the buffer.

Modification of the source hit frequency (by a factor $f$) could then be modelled by multiplication of the time of arrival of a hit by a constant $\frac{1}{f}$. Nevertheless, this approach does not simulate the high hit frequency sources well. By 'squeezing' all of the hits together by a factor, we also decrease the timespan of each cluster, which is does not occur in real data sources with high hit rate. This would decrease the number of clusters that were divided by the border, possibly leading to overly optimistic estimate of the performance. Therefore, when modifying the frequency, we first form the clusters, and offset only the first hit $h_{\text{first}}$ of each cluster by a factor $f$. And only then we change the time of arrival of remaining hits $h$ in the cluster, such that the relative time difference $h.\text{ToA} - h_{\text{first}}.\text{ToA}$ is preserved (thus preserving the original cluster timespan).

Even though we tried to simulate the online setting accurately, there is a factor we decided not to incorporate into the benchmarks. When the hits arrive from the chip, they arrive to the reader at a specific time. This effect could be simulated by employing a timer that would release the hits periodically. We decided against the timer (we release the hits at full speed, making a no-timer simulation) mostly because of two reasons. First, the timer provides additional computational overhead, which would not be present in the real-time scenario, possibly making the simulation less accurate. And second, we can estimate the performance of our models in real-time hit processing by modifying the fully online scenario in the way described below.

Modification: While receiving the hits, collect the hits in a container (as they arrive in real-time) and, after a fixed time, periodically release them to the rest of the computational graph at full speed.

Evidently, the fully online scenario must be at least as fast as the modification (they do the same work, but there is no waiting in the fully online approach). On the other hand, when comparing the modification to the no-timer simulation, the modification only introduces a fixed-time lag to the whole processing (other than the small time lag, there should be no difference).

We have shown that the maximal speed of real-time clustering should be at least as high as the modification, and the modification is almost as fast as the no-timer simulation (apart from a small fixed-time lag). Together the maximal speed of the no-timer approach is expected not to be notably faster than the fully real-time scenario.

One more effect that we did not incorporate into our simulation is the fact that the unorderedness changes as a function of hit frequency. Fortunately, the upper bound $t$ of the $t$-orderedness is still preserved ($t \approx 500$ $\mu$s but the hits are likely to arrive more temporally disordered. Again, we do not expect this fact to be significant, as sorting the data stream is not a bottleneck of the processing and has guaranteed $O(\log n)$ worst-case time complexity for each received hit ($n$ represents the number of hits obtained during time $t$).

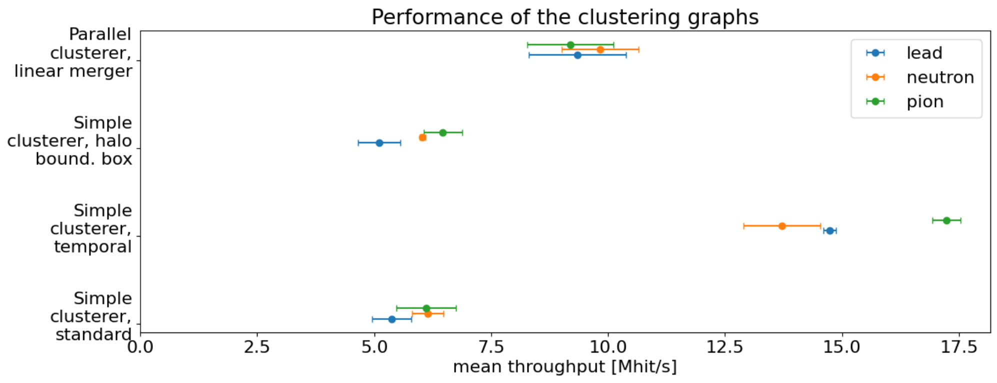
### 6.3.3 Results

In this subsection, we will provide the benchmarking results for architectures and models that we tested. Values of the parameters, if not stated otherwise:

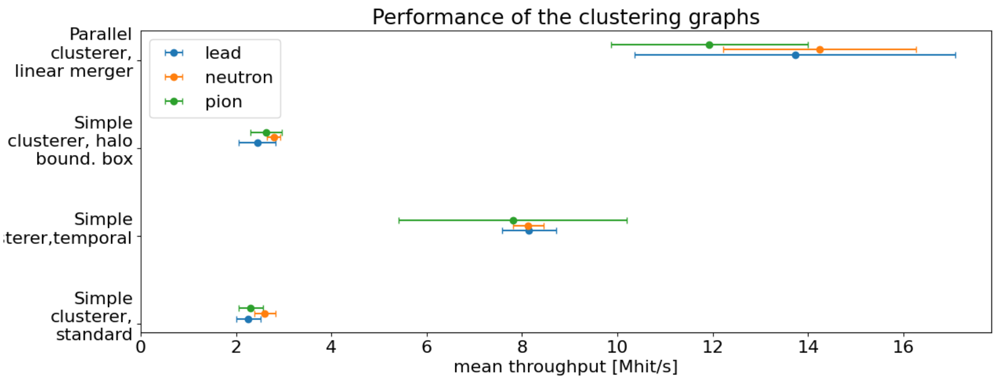
- $t_{\text{window}}$ – We used the value 8000 ns for all of the benchmarks.
- $n_{\text{workers}}$ – The default value of this parameter is 16 for the server and 4 for the laptop.
- Trigger – By default no trigger is used.
- Split node – The first 'reader' node is used as the node of the split.

#### Model comparison

Firstly, to get an estimate of the performance of some of the implemented models, we compared the performance of the approximative models with simple clustering and parallel clustering. We do not see any speed improvement for the halo clustering, which we assume is because the management of the bounding boxes and small halo size outweighed the potential speed gain. On the other hand, the temporal clustering is notably faster, which implies that the spatial neighbor checking in the standard clustering is the most computationally expensive part of the algorithm. Regarding the parallelization speedup, for the laptop with only 4 CPU cores, we were able to increase the throughput from 5-6 MHit/s to 9-10 MHit/s. On the server, however, the benefit from the parallelization was even higher (up to 14-15 MHit/s), as the machine has 16 CPU cores. Notably, the single core frequency on the server was lower, yielding lower performance for the computational graphs with a low degree of parallelization.



(a) Laptop benchmark

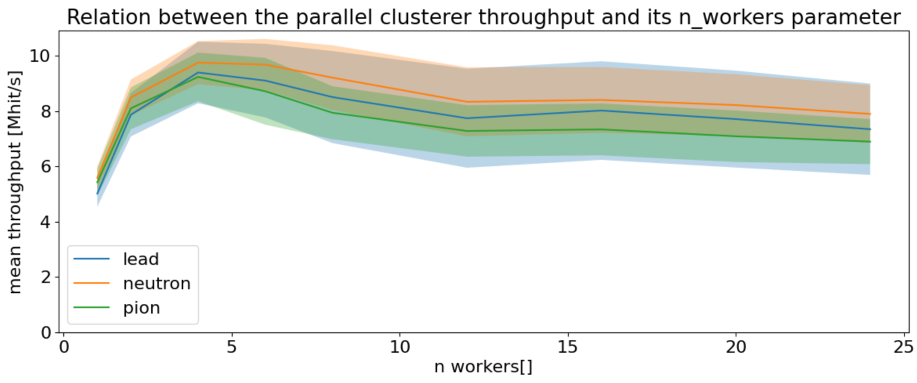


(b) Server benchmark

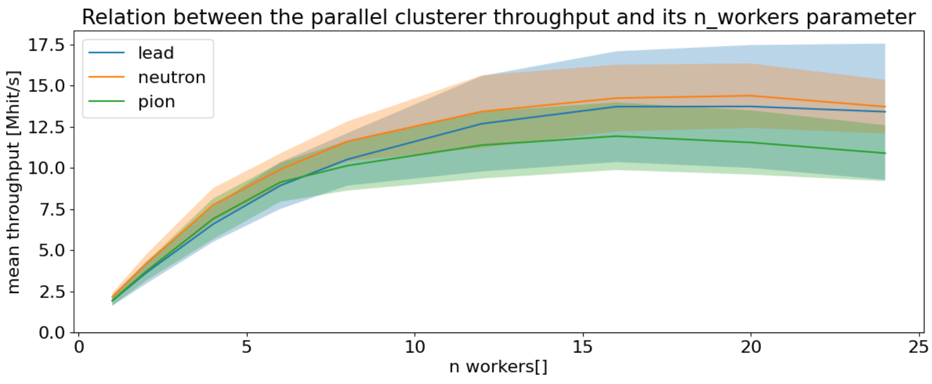
Figure 6.3: Throughput of the approximation clustering methods compared with the simple and parallel clustering.

### $n_{\text{workers}}$ parameter

Then we tested the influence of $n_{\text{workers}}$ parameter on the performance of the parallelization. Results are shown in Figure 6.3. For the laptop, we can see that the throughput peeks around the value $n_{\text{workers}} = 4$, which is the number of available cores in the CPU. Similarly, for the server, the maximum parallelization throughput was reached around the $n_{\text{workers}} = 16$. For higher values of $n_{\text{workers}}$, we can see a small decline in throughput which we attribute to the overhead which stems from the increasing size of the computational graph.



(a) Laptop benchmark

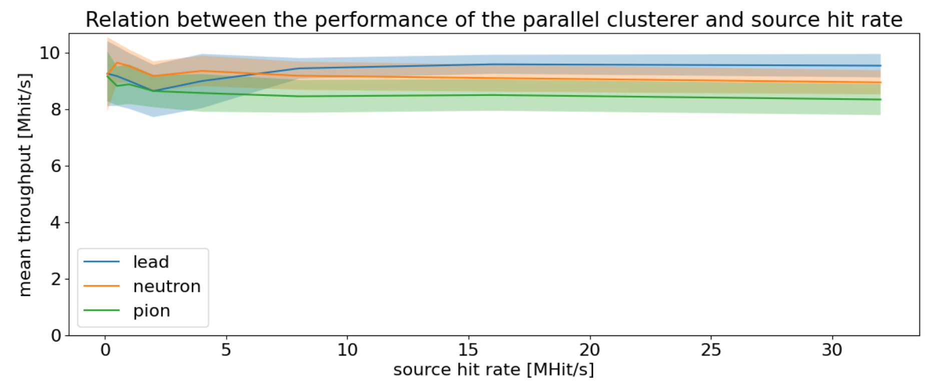


(b) Server benchmark

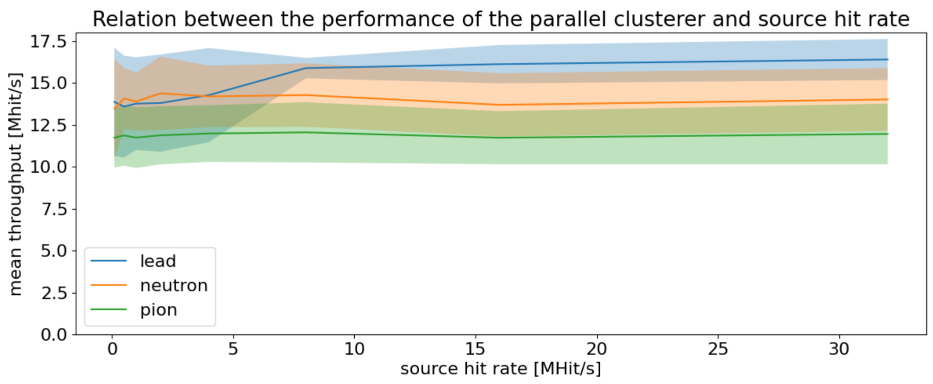
Figure 6.4: Dependence of the throughput of the parallel clustering on the $n_{\text{workers}}$ parameter (data-based split).

### Hit rate parameter

Then we decided to observe the parallelization performance of the algorithm for data with various hit rates (hit frequencies). Naturally, the observed hit rate is rarely constant, so instead of analyzing the average hit rate over the whole acquisition, we measure the hit rate for small time windows ($t \approx 200$ ms) and then take the maximum hit rate over the whole acquisition ($f = \max\{\frac{n_i}{t} | i \in \{1, 2, \ldots \frac{t_{\text{total}}}{t}\}\}$ with $n_i$ denoting the number of hits observed in $i$-th time window and $t_{\text{total}}$ representing the total acquisition time). The reason behind this is that we would like to measure how the clustering behaves under the given maximum hit rate. For instance, with higher hit rates, the number of clusters open at the time grows, which could potentially slow down the clustering. During the sorting phase, we also have to keep more hits in the memory at each time frame. However, from the experiment (see Figure 6.5), we can see that both on the laptop and the server, modifying the hit rate does not have a significant negative impact on the overall performance.



(a) Laptop benchmark

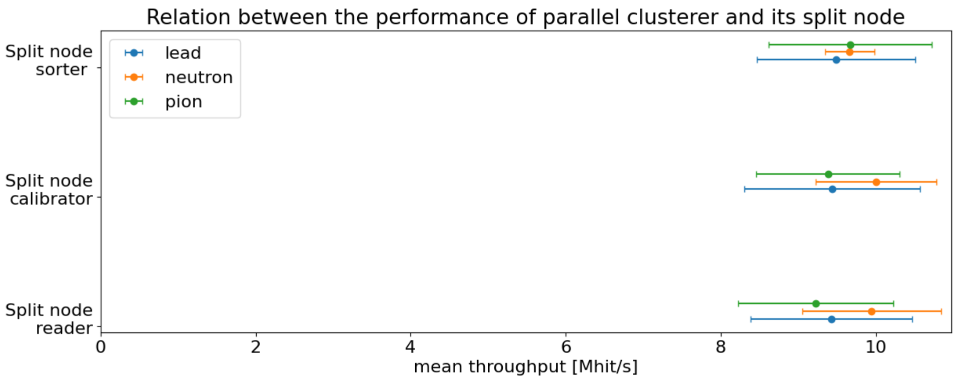


(b) Server benchmark

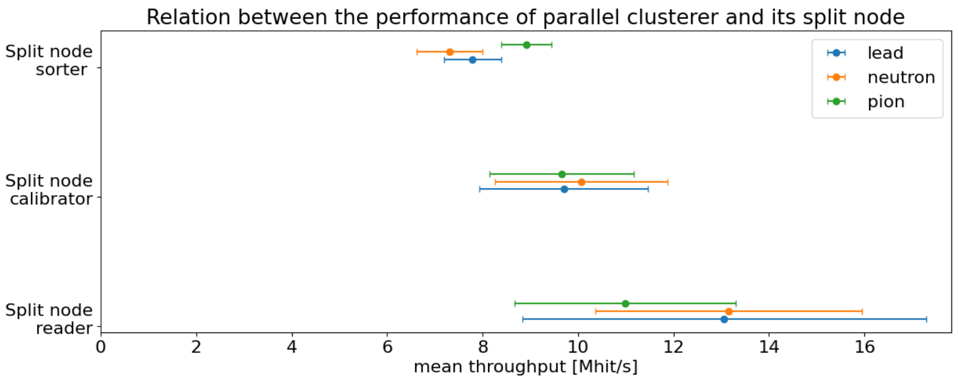
Figure 6.5: Dependence of the throughput of the parallel clustering on the frequency of the hit source.

### Split node

Another factor we decided to test was the node, where the data-based split was performed. We provided three options – the reader node, the calibrator node, and the sorter node (see Figure 6.6). Here we see a difference between the laptop and the server. On the laptop, because of a lower degree of parallelization, the earlier data splitting does not improve the throughput. On the contrary, the server was able to capitalize on the spit as early as in the reader, providing a noticeable improvement when compared to the splits later in the graph.



(a) Laptop benchmark

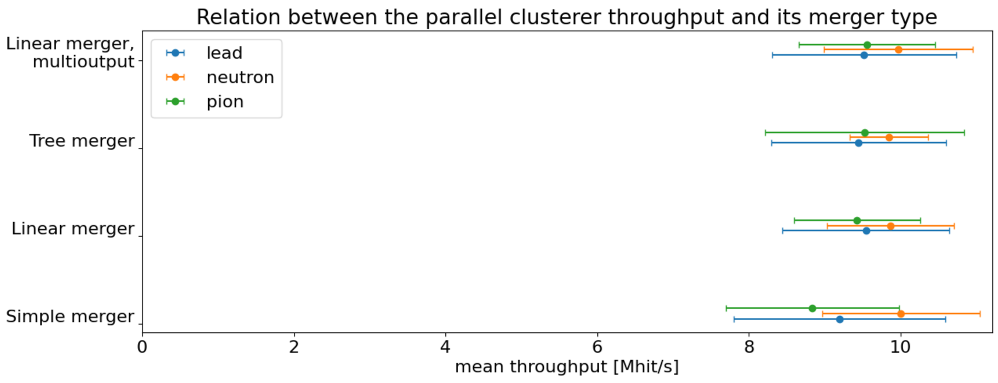


(b) Server benchmark

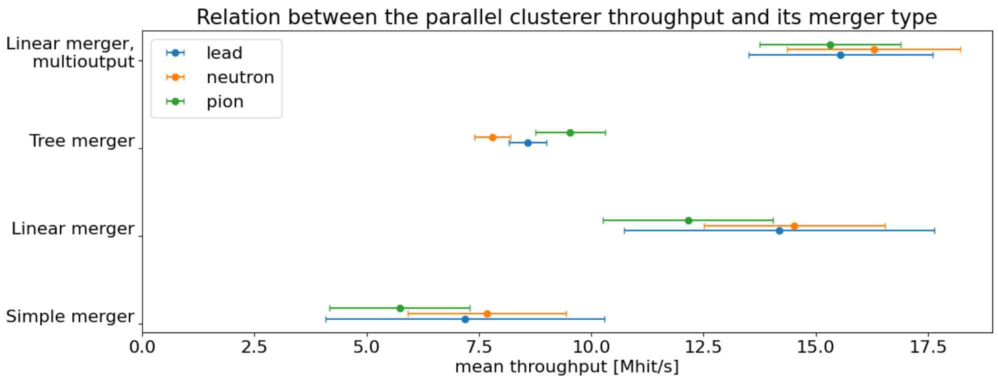
Figure 6.6: Dependence of the clusterer throughput on the node where the (data-based) splitting occurs.

## Merger type

In order to assess the effectiveness of the proposed merging methods we also included them in our benchmarks (see Figure 6.7). On the laptop, the difference between the methods was not really significant. But with the higher degree of parallelization, we can see that linear (parallel) merging is faster than the single-worker approach. Additionally, it is observed that if we keep the parallel streams separated (multioutput where all streams contain complete, already merged clusters) and output them separately, we can observe another performance gain (in total, more than 15 MHit/s).



(a) Laptop benchmark

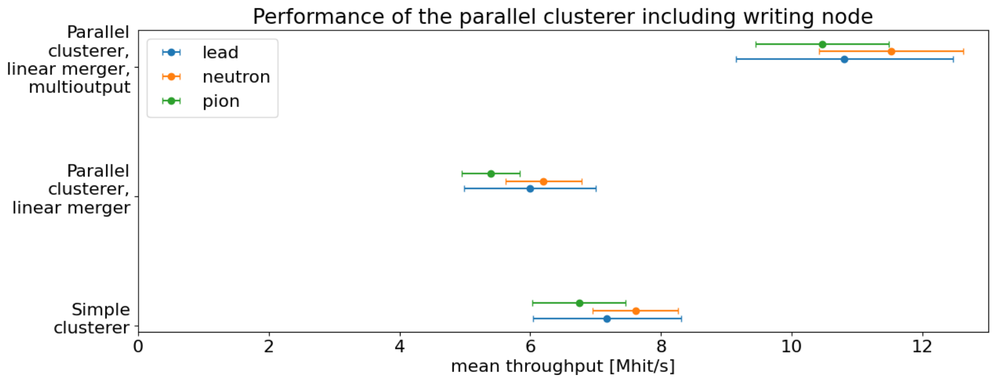


(b) Server benchmark

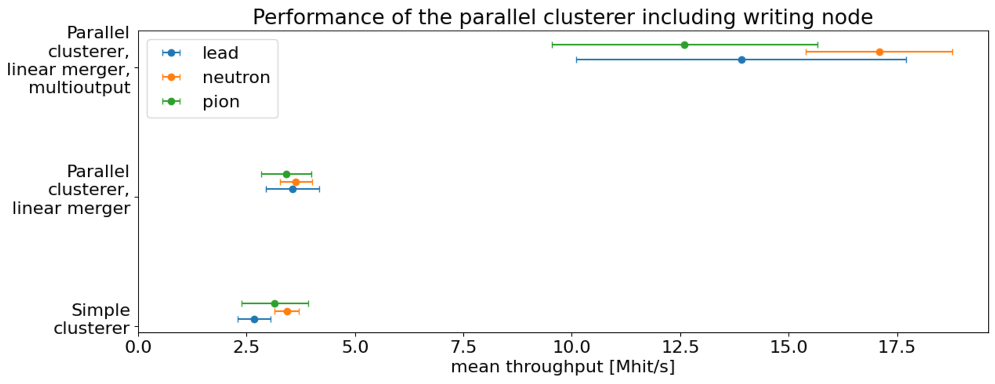
Figure 6.7: Dependence of the parallel clustering throughput on type of merging used.

### Writing

Finally, we decided to include also writing into a file (I/O) in the benchmark (Figure 6.8). On both the server and the laptop, we can observe a decrease in the throughput for the models where the data streams are eventually merged (the parallel clustering was slowed down from 14 MHit/s to approximately 4 MHit/s). In these computational graphs, only a single node is responsible for the writing of the clusters. This might have one of the two explanations. Either the hardware is not capable of writing at a faster rate, or we did not spend enough computational resources on the writing itself. By observing the benchmark of the multioutput graph, we can quickly see that the latter is the case. For the applications where parallel writing is supported, we were able to preserve the mean clustering speed at 10 Mhit/s for the laptop and almost 15 Mhit/s for the server. Notably, on average, each hit occupies around 30 characters in a file, which translates to 30 bytes in UTF-8 encoding (we only use ASCII characters). Thus, 15 Mhit/s means writing 450 MB/s which might not be far from the hardware limit of the SSD on the server.



(a) Laptop benchmark



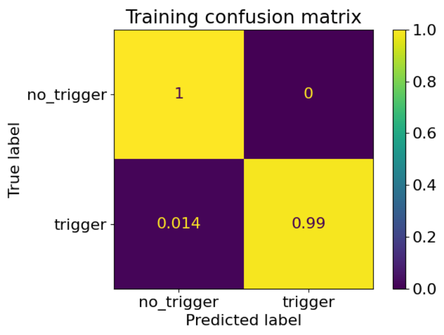
(b) Server benchmark

Figure 6.8: The throughput of the models with included I/O operations for writing. It is evident that parallel writing provides a noticeable performance boost.

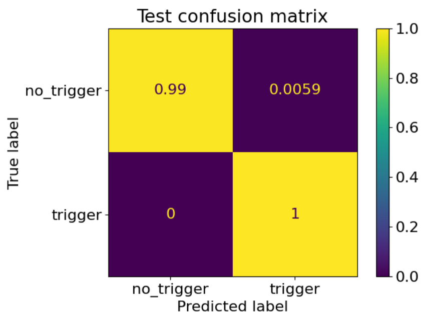
### ML trigger clustering

To show that the triggers can actually be helpful in some scenarios, we decided to show examples of how they can be used. We will try to selectively cluster in the environment with a recurring beam with a high particle flux. The beam repeatedly starts and stops for $\approx 5$ second intervals. We can try to detect these changes (when the beam starts and stops) and only run clustering for a short while after the change occurs. This way, if the particle field does not change significantly, no data need to be clustered.

In order to detect the changes in the field, we can use the differentiated features (see Section 5.2.1). We automatically select the windows where the beam change occurred according to the corresponding change in the hit rate attribute. These windows represent the 'trigger' class. The remaining windows belong to the 'no_trigger' class. In the experiment, we used a window of size 200 ms, and the after activating, the trigger remained active for 500 ms.



(a) Confusion matrix evaluated on the training set for the trained MLP.



(b) Confusion matrix evaluated on the test set for the trained MLP.

Figure 6.9: Confusion matrices for trained MLP model. The confusion matrices for SVM are not shown as they do not differ significantly from the confusion matrices above.

### Class imbalance

Because of the class imbalance, we decided also to employ oversampling. The examples of the minority class are copied multiple times, so they significantly contribute to the loss function. However, such a simple oversampling can easily cause the model to overfit. Because of that, during the oversampling, we added Gaussian noise $N(\mu_{\text{noise}}, \sigma_{\text{noise}})$ to each attribute of the copied sample. We chose $\mu_{\text{noise}} = 0$. To set $\sigma_{\text{noise}}$, we computed the true standard deviation of the dataset $\sigma_{\text{true}}$ for each attribute. Then we set $\sigma_{\text{noise}} = c \cdot \sigma_{\text{true}}$ for a modifiable constant $c$ (for example $c = 0.1$).

### Training

For the detection of the start/stop of the beam, we trained MLP and SVM. Both models can be trained sufficiently for this relatively simple task as shown in Figure 6.9.

### Application

After applying the trained trigger to the data from pion measurements, we observed that both triggers discarded approximately 67% of all processed hits. On the laptop, this resulted in an increase in throughput for MLP trigger up to 16.3 MHit/s and for SVM trigger up to 17.5 MHit/s. On the server, however, the performance gain was not as significant, reaching ‘only’ 11.8 MHit/s for MLP and 14.2 MHit/s for the SVM trigger. However, the maximum clustering speed is not the only criterion to keep in mind. Besides the possible speedup, the triggers also reduce the data.

### Outlier detection

Apart from supervised learning models, we also decided to demonstrate an example usage of an unsupervised learning method. The one-class SVM model will detect the outliers in the data and only perform clustering on the ‘irregular’ windows. This can be used to search for exotic particles. After the training, the model discarded 89% of the received hits, which increased the maximum throughput to 26 MHit/s on the laptop and 17 MHit/s on the server. The remarkable difference between the throughputs we attribute mostly to the difference in the single-core CPU frequency between the devices.

# Conclusion

The main focus of the thesis was to explore options for the possible acceleration of the clustering process and the related data reduction in the data obtained from the Timepix3 detector. The summary of the achieved results is listed below.

## Clustering parallelization

First, we split the clustering process logically into multiple steps (I/O, time sorting, calibration,. . . ) and performed each step in parallel, together forming a pipeline. Then, we created multiple such pipelines and split the data between those based on the time of arrival. Nevertheless, the splitting of the hits between the clusters may create incomplete clusters, so additional merging was needed, which was also parallelized.

## Approximative clustering

Second, we implemented two ways of approximative clustering – 'tiled clustering' and 'halo bounding box clustering'. The first uses the idea to consider pixel neighborhoods of larger tiles instead of individual pixels in order to overcome the potential dead (not-activated) pixels. The 'halo bounding box' idea was utilize the phenomenon known as the halo effect to simplify neighborhood search during clustering. If we register two hits not far from each other both temporally and spatially (not necessarily neighboring), and one of the pixels has high deposited energy, these hits likely belong to the same cluster. Besides that, we also decided to resign on the spatial neighborhood of the pixel during clustering and named this approach 'temporal' clustering. All mentioned methods are compatible with the parallelization above and can be used within any of the parallel architectures.

## Selective (machine-learning-based) trigger clustering

And third, we provided software for the creation of clustering triggers, which initiate clustering with the signs of interesting events. For each time window, we compute features (or their temporal difference) to obtain a lower dimensional representation of the hits which occurred in the time frame. Then, the events of interest can be characterized explicitly or learned from the data using machine-learning methods. This has the capability to reduce the data which would be otherwise stored on a disk.

### Evaluation and validation

And last, our methods were tested using simulated and real-life datasets. We validated the clustering models against an existing clustering approach. Additionally, from the experiments, we conclude that parallelization significantly accelerates the clustering, scaling with the available number of threads. For the laptop with four cores (4GHz frequency), we achieved maximum throughput of around 9 MHit/s. On a server with sixteen cores (each 2GHz frequency), we were able to reach the throughput of 14 MHit/s. The speeds were originally affected by the writing I/O operations, but with parallel multifile writing, we were able almost to preserve the mentioned throughput values, corresponding to the data rate of more than 400 MB per second (30 bytes per hit). With the halo-based clustering, we do not see any significant improvement – we attribute it to the extra computational cost of management of bounding boxes. On the other hand, the temporal clustering increased the maximal throughput, but as we have shown, its correctness decays with increasing data rates. However, temporal clustering is much less computationally expensive, which can be useful in space, where the resources are extremely costly.

For the trigger clustering, we used differential window features to find the rising and falling edge in the environment where the main radiation source is repeatedly turned on and off. The perceptron and support vector machine models were trained to detect this edge and reduced the data by more than a half, which led to an increase in maximal throughput for the laptop to 16 Mhit/s. For the server, the maximal throughput remained at 12-14 Mhit/s, possibly suffering from the lower single-core frequency. Additionally, we also trained an unsupervised model for outlier detection – a one-class support vector machine. In the simple experiment, it discarded almost 90% of the data, while increasing the maximum throughput to 26 Mhit/s on the laptop and 17 MHit/s on the server.

As our main contribution, we consider the parallelization which increased the clustering throughput in comparison with the baseline method (performing at around 3 Mhit/s) up to 10-15 MHit/s, depending on the hardware used. To sum it up, we consider all our goals to be completed.

## Future work

Even though we believe we have fulfilled our goals, there is still room for possible improvements.

- **Two phase clustering** – one of the promising clustering methods which was not implemented, is the idea of splitting the clustering into temporal and spatial steps. The temporal clustering serves as a preprocessing step, and in the spatial step, the cluster is potentially split into multiple clusters using some graph component partitioning algorithm (BFS, DFS, . . . ). In principle, it could save time, as no cluster merging takes place, and no complex auxiliary data structures are required.

- **Sorting first and dynamic time window** – In our text we argued, that dynamic time windows are not feasible for the setup we use. Nevertheless, we could modify it and first perform the time sorting in parallel, using round-robin splitting and merge the sorted streams. Then when the data is sorted, it can be splitted again (for calibration, clustering and cluster merging), using a splitting time window that can change in time (smaller window for high-frequency data sources) to guarantee equal distribution of work in every case. The speed improvement is not guaranteed as the merging of the sorted hit streams could become the new bottleneck.
- **Parameter tuning** – In our work, we examined the influence of a few parameters on the clustering but their internal dependence was not tested. For instance, a grid search algorithm can be used to find the optimal parameters for the specific computational architecture.
- **Trigger parallelization** – We decided to perform the trigger computation in a single node (the clustering is still done in parallel) to avoid synchronization between nodes. Nevertheless, the cost of synchronization still remains to be experimentally determined.
- **Online outlier detection** – An improvement of the trigger clustering could be to develop a fast algorithm for outlier detection, which would be trained at the beginning, but even during the inference phase it would periodically try to update its knowledge base about the data to online detect the changes in the radiation field.

# Bibliography

[1] T Poikela, J Plosila, T Westerlund, M Campbell, M De Gaspari, X Llopart, V Gromov, R Kluit, M Van Beuzekom, F Zappon, V Zivkovic, C Brezina, K Desch, Y Fu, and A Kruth. Timepix3: a 65K channel hybrid pixel readout chip with simultaneous ToA/ToT and sparse readout. *Journal of Instrumentation*, 9(05):C05013–C05013, May 2014.

[2] Rafael Ballabriga, Michael Campbell, and Xavier Llopart. Asic developments for radiation imaging applications: The medipix and timepix family. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 878:10–23, 2018. Radiation Imaging Techniques and Applications.

[3] J Jakůbek. Semiconductor pixel detectors and their applications in life sciences. *Journal of Instrumentation*, 4(03):P03013, mar 2009.

[4] B. Bergmann, T. Billoud, P. Burian, C. Leroy, P. Mánek, L. Meduna, S. Pospíšil, and M. Suk. Particle tracking and radiation field characterization with Timepix3 in ATLAS. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 978:164401, October 2020.

[5] M. Martišíková, T. Gehrke, S. Berke, G. Aricò, and O. Jäkel. Helium ion beam imaging for image guided ion radiotherapy. *Radiation Oncology*, 13(1):109, December 2018.

[6] X. Llopart, R. Ballabriga, M. Campbell, L. Tlustos, and W. Wong. Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 581(1-2):485–494, October 2007.

[7] Carlos Granja, Stepan Polansky, Zdenek Vykydal, Stanislav Pospisil, Alan Owens, Zdenek Kozacek, Karim Mellab, and Marek Simcak. The SATRAM Timepix spacecraft payload in open space on board the Proba-V satellite for wide range radiation monitoring in LEO orbit. *Planetary and Space Science*, 125:114–129, June 2016.

[8] X. Llopart, J. Alozy, R. Ballabriga, M. Campbell, R. Casanova, V. Gromov, E.H.M. Heijne, T. Poikela, E. Santin, V. Sriskaran, L. Tlustos, and A. Vitkovskiy. Timepix4, a large area pixel detector readout chip which

can be tiled on 4 sides providing sub-200 ps timestamp binning. *Journal of Instrumentation*, 17(01):C01044, January 2022.

[9] Declan Garvey. Pokročilé metody dekompozice radiačního pole hybridními pixelovými detektory, June 2023.

[10] Florian Michael Pitters, Andreas Matthias Nurnberg, Magdalena Munker, Dominik Dannheim, Adrian Fiergolski, Daniel Hynds, Xavi Llopart Cudie, Niloufar Alipour Tehrani, Morag Jean Williams, Wolfgang Klempt, and others. Time and energy calibration of timepix3 assemblies with thin silicon sensors. Technical report, 2018.

[11] P. Burian, P. Broulím, M. Jára, V. Georgiev, and B. Bergmann. Katherine: Ethernet Embedded Readout Interface for Timepix3. *Journal of Instrumentation*, 12(11):C11001–C11001, November 2017.

[12] Lukáš Meduna. Detecting elementary particles with Timepix3 detector, June 2019. Accepted: 2021-03-25T23:04:58Z Publisher: Univerzita Karlova, Matematicko-fyzikální fakulta.

[13] Lukáš Meduna, Benedikt Bergmann, Petr Burian, Petr Mánek, Stanislav Pospíšil, and Michal Suk. Real-time timepix3 data clustering, visualization and classification with a new clusterer framework, 2019.

[14] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[15] Petr Mánek. A system for 3D localization of gamma sources using Timepix3-based Compton cameras, September 2018. Accepted: 2018-10-01T12:23:27Z Publisher: Univerzita Karlova, Matematicko-fyzikální fakulta.

[16] G.T. Byrd and M.J. Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 87(3):456–466, March 1999.

[17] William S Noble. What is a support vector machine? *Nature Biotechnology*, 24(12):1565–1567, December 2006.

[18] S Hoang, R Vilalta, L Pinsky, M Kroupa, N Stoffle, and J Idarraga. Data Analysis of Tracks of Heavy Ion Particles in Timepix Detector. *Journal of Physics: Conference Series*, 523:012026, June 2014.

[19] C. Gemme. The ATLAS pixel detector. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 501(1):87–92, March 2003.

[20] Wesley H Smith. Triggering at LHC experiments. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 478(1-2):62–67, February 2002.

[21] St. Gohl, M. Malich, B. Bergmann, P. Burian, C. Granja, E. Heijne, M. Holik, J. Jacubek, J. Janecek, L. Marek, C. Oancea, M. Petro, S. Pospisil, A. Smetana, P. Soukup, D. Turecek, and M. Vuolo. A miniaturized radiation monitor for continuous dosimetry and particle identification in space. *Journal of Instrumentation*, 17(01):C01066, jan 2022.

[22] Alvin W. Moore and James W. Jorgenson. Median filtering for removal of low-frequency background drift. *Analytical Chemistry*, 65(2):188–191, January 1993.

[23] Fionn Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5-6):183–197, July 1991.

[24] Allan Pinkus. Approximation theory of the MLP model in neural networks. *Acta Numerica*, 8:143–195, January 1999.

[25] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[26] Shan Suthaharan. Support Vector Machine. In *Machine Learning Models and Algorithms for Big Data Classification*, volume 36, pages 207–235. Springer US, Boston, MA, 2016. Series Title: Integrated Series in Information Systems.

[27] Shirish K Shevade, S Sathiya Keerthi, Chiranjib Bhattacharyya, and Karaturi Radha Krishna Murthy. Improvements to the smo algorithm for svm regression. *IEEE transactions on neural networks*, 11(5):1188–1193, 2000.

[28] Maryamsadat Hejazi and Yashwant Prasad Singh. ONE-CLASS SUPPORT VECTOR MACHINES APPROACH TO ANOMALY DETECTION. *Applied Artificial Intelligence*, 27(5):351–366, May 2013.

[29] Mennatallah Amer, Markus Goldstein, and Slim Abdennadher. Enhancing one-class support vector machines for unsupervised anomaly detection. In *Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description*, pages 8–15, Chicago Illinois, August 2013. ACM.

[30] Anjana Gosain and Saanchi Sardana. Handling class imbalance problem using oversampling techniques: A review. In *2017 International Conference*

*on Advances in Computing, Communications and Informatics (ICACCI)*, pages 79–85, Udupi, September 2017. IEEE.

[31] Gustavo E. A. P. A. Batista, Ronaldo C. Prati, and Maria Carolina Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations Newsletter*, 6(1):20–29, June 2004.

[32] Sebastian Raschka. An overview of general performance metrics of binary classifier systems. *arXiv preprint arXiv:1410.5330*, 2014.

[33] Ayush Shridhar, Phil Tomson, and Mike Innes. Interoperating deep learning models with onnx. jl. In *Proceedings of the JuliaCon Conferences*, volume 1, page 59, 2020.

[34] Sam Fletcher and Md Zahidul Islam. Comparing sets of patterns with the Jaccard index. *Australasian Journal of Information Systems*, 22, March 2018.

[35] P.A. Larson. External sorting: run formation revisited. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):961–972, 2003.

# List of Figures

# List of Tables

# A. Attachments

## A.1 User manual for the developed tools

This chapter is dedicated to the description of the implemented tools for the clustering, proposed in the thesis above. It consists of three programs:

- *clusterer*,
- *window_processor* and
- *window_trigger_creator*.

Each of the programs has a dedicated section where we discuss the installation and usage of the program.

Currently, for the clusterer and window_processor programs, there are two options for installation – either to build it from the source, or to download its pre-built executable. For the window_trigger_creator, we provide only the build option. Distribution via a single executable would need to contain all dependencies, including large machine-learning libraries, which would take a lot of space (more than a gigabyte).

- Build from source:
  - `https://github.com/TomSpeedy/clusterer`
  - `https://github.com/TomSpeedy/window_processor`
  - `https://github.com/TomSpeedy/window_trigger_creator`

  Properties:

  + Wider support of various operating systems.

  + Possibility of extending the source code.

  – Requires more time to set up dependencies.

- Download pre-built executables (`https://drive.google.com/drive/folders/1DOCHbyQ9w5nK9EDZub-YpGhHROj_7BhL?usp=sharing`).

  Properties:

  + Fast and simple.

  – Currently only supports Ubuntu OS and 64-bit Windows.

For the purpose of testing all three applications, we share a (relatively) small dataset, see the link `https://drive.google.com/drive/folders/1DOCHbyQ9w5nK9EDZub-YpGhHROj_7BhL?usp=sharing`. In case of any problems or questions, please send an email to ‘celko.tom@gmail.com’.

## A.1.1 Clusterer

The clusterer is a command-line tool meant for the first step of event building called clustering. As of now, the clusterer works in the offline mode, processing hits obtained from Timepix3 in a text file format.

### Installation

- From pre-built binary – download the zipped folder and simply run the executable. (named ‘clusterer’ or ‘clusterer.exe’)
- Build from source:

  Prerequisites:

  1. OnnxRuntime (version for Ubuntu is distributed within the clusterer repository).
  2. Cmake version $\geq 2.8$.
  3. C++ compiler with support of C++17 standard. (GCC version 8+, Visual Studio 2017 15.8 – MSVC 19.15+ or similar).

  Note: Steps marked by * are not required for Linux systems.

  1. Clone the repository using HTTPS or SSH.
  2. *Download OnnxRuntime from `https://onnxruntime.ai/`. Select ‘optimize for inference’ and choose the target platform. Choose ‘C++’ API and in ‘hw acceleration’ field choose ‘default cpu’. Follow the instructions for installation.
  3. Create the build directory (‘clusterer/build/’) and navigate to it.
  4. Run **`cmake ..`**, this generates the build files. For the MSVC compiler, this generates VS solution. For this solution, the Onnx runtime needs to be added in the form of a NuGet package.
  5. *Set environmental variable‘ONNX_LIBRARY_PATH’ to the directory where the onnxruntime library is located (search for a file named ‘onnxruntime.dll’, ‘onnxruntime.so’ or similar).

6. Run `cmake --build ..` For the MSVC it is required to add `--config Release` as the default build is Debug. The executable is then located in the 'Release' or 'bin' folder. If the build does not succeed, observe the error message, it might point toward the problem. Alternatively, contact the author.

### Usage

In order to use the clusterer, the user is expected to provide command line arguments. To display the options pass the `--help` argument. The arguments and the options are always separated by a blank space. The clusterer can be run in multiple configurations, specified by the 'mode' option.

- `benchmark` – To run this option, please download the test dataset from the link `https://drive.google.com/drive/folders/1DOCHbyQ9w5nK9EDZub-YpGhHROj_7BhL?usp=sharing` and extract it. This option requires only a single additional argument – path to the downloaded dataset.
- `window_processor` – This option serves to generate file with unclustered window features, further used in the window processor app. Additionally, the calibration folder should be specified using `--calib` option.
- `compare` – The 'compare option is a tool for comparison of two clustered files. It computes the Intersection over union match between the cluster datasets.
- `clustering` – The default mode, which clusters the given data file. Similarly to window_processor, the calibration folder should be provided.

The arguments for the program are passed in two ways. Some of the arguments are passed directly via the command line. Additional 'Node arguments' for the computational nodes are passed in a file via `--args` argument. All options and arguments:

- `--mode` <one of 'clustering', 'window_features', 'compare', 'benchmark'>
  – defaults to 'clustering'
  – mandatory argument if 'clustering' or 'window_features': `--calib`
- `--help`
- `--args` <path_to_node_argument_file>– a file with a set of parameters for each node in the computational graph
- `--calib` <path_to_calibration_folder>

- `--merge_type` <one of 'none', 'simple', 'single_layer', 'tree' or 'multioutput'>
  – defaults to 'single_layer' – chooses the type of merging the datastreams, if 'none' is chosen, the no data-based split takes place, and 'n_workers' parameter is ignored

- `--output` <path_to_output_folder>
  – a directory where the output is written
  – defaults to <binary_location>/../../output

- `--debug`
  – prints extra information about the dataflow in the computational graph, which can be useful for debugging purposes

- `--n_workers` <positive integer>
  – the width of the data-based split for the clustering architecture
  – defaults to 4

- `--clustering_type` <one of 'standard', 'temporal', 'tiled', 'halo_bb'>
  – defaults to 'standard'

**Node argument file**

In 'node arguments' configuration file, the user can override the default parameters of each node by specifying the `--args` argument file. The file has the following structure:

node_name
–[node_property1_name]:[node_property1_value]
–[node_property2_name]:[node_property2_value]
another_node_name
. . .

Currently, the following node names and properties are supported:

- reader
  - split:['true' or 'false']
    Sets if the reader node should perform the split for parallel clustering computation if 'merge_type' is 'none'. The first node (in the order of the data flow) with this flag set to 'true' is the split node. The default is 'true'.
  - sleep_duration_full_memory:[positive real] Sets the duration for which the reader is inactive after the memory limit is reached (in microsec-

onds). After this time, if the memory check succeeds, the processing is restored. Defaults to 100.

- calibrator
  - split:[‘true’ or ‘false’]
    Analogically, see the same option for reader node.
- sorter
  - split:[‘true’ or ‘false’]
    Analogically, see the same option for reader node.
- clusterer
  - max_dt:[positive real]
    The maximum time in nanoseconds for hits to be considered ‘temporally neighboring’. During clustering, two hits $A$ and $B$ belong to a single cluster if there exists a path of hits where each hit on the path is spatially and temporally neighboring. It defaults to 200.
  - tile_size:[one of 1, 2, 4, 8, 16]
    If set to a value $d > 1$, the detector is split into tiles of size $d \times d$ pixels. When checking the spatial 8-neighborhood of a pixel, all pixels in the neighborhood of a tile are considered as neighboring to this pixel. This can effectively ignore the ‘dead’ pixels. The default is 1.
- trigger
  - use_trigger:[‘true’ or ‘false’]
    An important variable, indicating if the trigger should be used. The default is ‘false’.
  - window_size:[positive real]
    The size of a time window (in nanoseconds) after which the window statistics are evaluated. **Please, use the same value which was used by the ‘window computer’ node.** Use with care; a small value of this parameter can significantly slow down the processing. The default is $2 \cdot 10^8$ ns (200ms).
  - diff_window_size:[positive real]
    The size of a time window (in nanoseconds) with respect to which the features are differentiated. After that, the median of these differences is chosen as the difference value. A value smaller than window_size implies no differentiation. Too high a value can notably slow down

the processing. **Please, use the same value which was used by the 'window computer' node.** The default is $1 \cdot 10^9$ ns (1 s), which corresponds to five default time windows.

  - trigger_time:[positive real]
    The trigger time is the time for which the trigger remains active after the initial activation. If the trigger reactivates during this time window, the active time of the trigger is extended adequately. The default is $5 \cdot 10^8$ (500 ms).
  - trigger_file:[a path to a trigger file]
    The trigger file path should point to the existing trigger file created by the 'window trigger creator' application. This trigger file is a (possibly machine-learning) model serialized in an ONNX format. The suffix is used to determine the correct type of the model – please do not modify (.nnt = neural network (from tensorflow package), .svmt = support vector machine, .osvmt = one-class support vector machine (from scikit-learn package)). The type of model needs to be assessed correctly, as some of the models produce multiple outputs of various data types. Alternatively, if the suffix is '.ift', the trigger file contains serialized intervals of features when the trigger should be active.

- halo_bb_clusterer
  - window_size:[positive real]
    The time window for hits that are clustered in the same buffer.
- window_computer
- window_size:[positive real]
  Analogical to the same property of the trigger node. It can be set via the 'window computer' GUI. The default is $2 \cdot 10^8$ (200ms).
- diff_window_size:[positive real]
  Analogical to the same property of the trigger node. It can be set via the 'window computer' GUI. The default is $1 \cdot 10^9$ (1s).

**Example use cases:**

- `./clusterer --calib [path/to/my/calib/folder/] [path/to/file/for/clustering]`
- `./clusterer --calib [path/to/my/calib/folder/] --args [path/to/node_args_file.txt] [path/to/file/for/clustering.txt]`

- `./clusterer --mode benchmark [/path/to/clusterer_data]`
- `./clusterer --mode compare [/path/to/first_clustered_file.ini] [/path/to/second_clustered_file.ini]`

### A.1.2 Window processor

#### Installation

- From pre-built binary – download the zipped folder and simply run the executable. (named 'window_processor' or 'window_processor.exe')
- Build from source:

  Prerequisites:

  1. OnnxRuntime (version for Ubuntu is distributed within the clusterer repository).
  2. Cmake version $\geq$ 2.8.
  3. C++ compiler with support of C++17 standard. (GCC version 8+, Visual Studio 2017 15.8+ – MSVC 19.15+ or similar).
  4. Build clusterer from the source. Apart from the executable, the build generates a clusterer library (.so,.dll,...), required by the window processor.

  Note: Steps marked by * are not required for Linux systems. After the prerequisites are matched, you may proceed further.

  1. Either place the window_processor project in the same directory as the clusterer, or preferably, set the environmental variable 'CLUSTERER_PATH' to the parent folder of the clusterer project.
  2. Proceed similarily to the build procedure of clusterer (as shown below).
  3. Create the build directory ('window_processor/build/') and navigate to it.
  4. Run `cmake ..`, this generates the build files. For the MSVC compiler, this generates VS solution. For this solution, the Onnx runtime needs to be added in the form of a NuGet package.
  5. *Set environmental variable 'ONNX_LIBRARY_PATH' to the directory where onnxruntime library is located (search for a file named 'onnxruntime.dll', 'onnxruntime.so' or similar).

6. Run `cmake --build ..` For the MSVC it is required to add `--config Release` as the default build is 'Debug'. The executable is then located in the 'Release' or 'bin' folder. If the build does not succeed, check the error message, it might point toward the problem. Alternatively, contact the author.

**Usage**

1. Select the input file. You can either:

   - type the path manually,
   - click 'Browse' and navigate to the file, or
   - **drag and drop** the file into the text field.

   The input file should be in the Burda-file format, where each line corresponds to a hit and consists of four integers: linear coordinate of the pixel, time of the arrival using the low-frequency clock, time of arrival using the high-frequency clock, and time over the threshold.

2. Select the calibration folder. The same three options are available as mentioned above.

3. Optionally set the differentiation window size **in milliseconds**.

4. Set the window size, also **in milliseconds**.

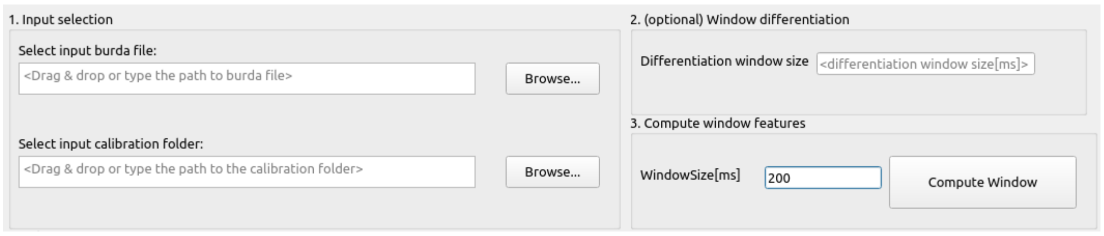
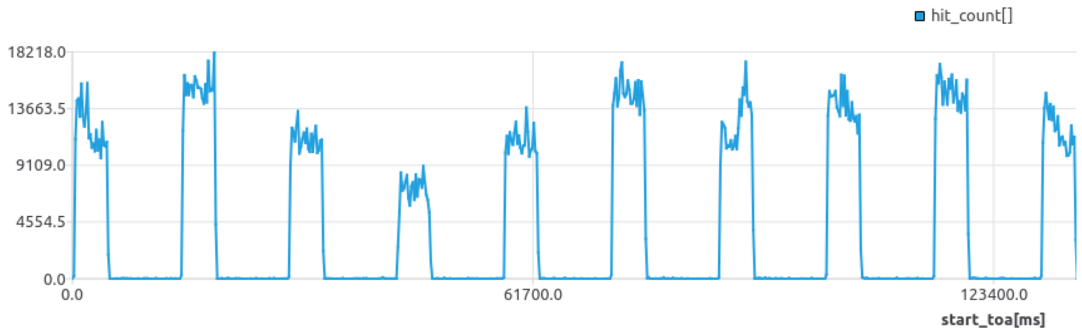

1. Input selection
Select input burda file:
<Drag & drop or type the path to burda file>
Browse...
Select input calibration folder:
<Drag & drop or type the path to the calibration folder>
Browse...
2. (optional) Window differentiation
Differentiation window size
<differentiation window size[ms]>
3. Compute window features
WindowSize[ms]
200
Compute Window

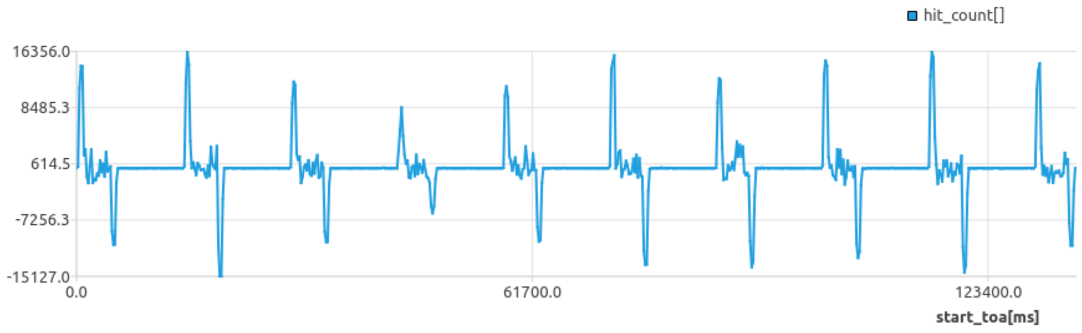
Figure A.1: Input selection fields, as described by the fields, as described by steps above.

5. Click 'Compute window'. The table with windows and its plots should be displayed.

6. Click on the column in the header of the table to display the attribute's plot with respect to the time below. Select the region on the x-axis to zoom in on the plot. Right-click on the plot to zoom out.



(a) Plot of non-differentiated feature (hit count) with respect to time.



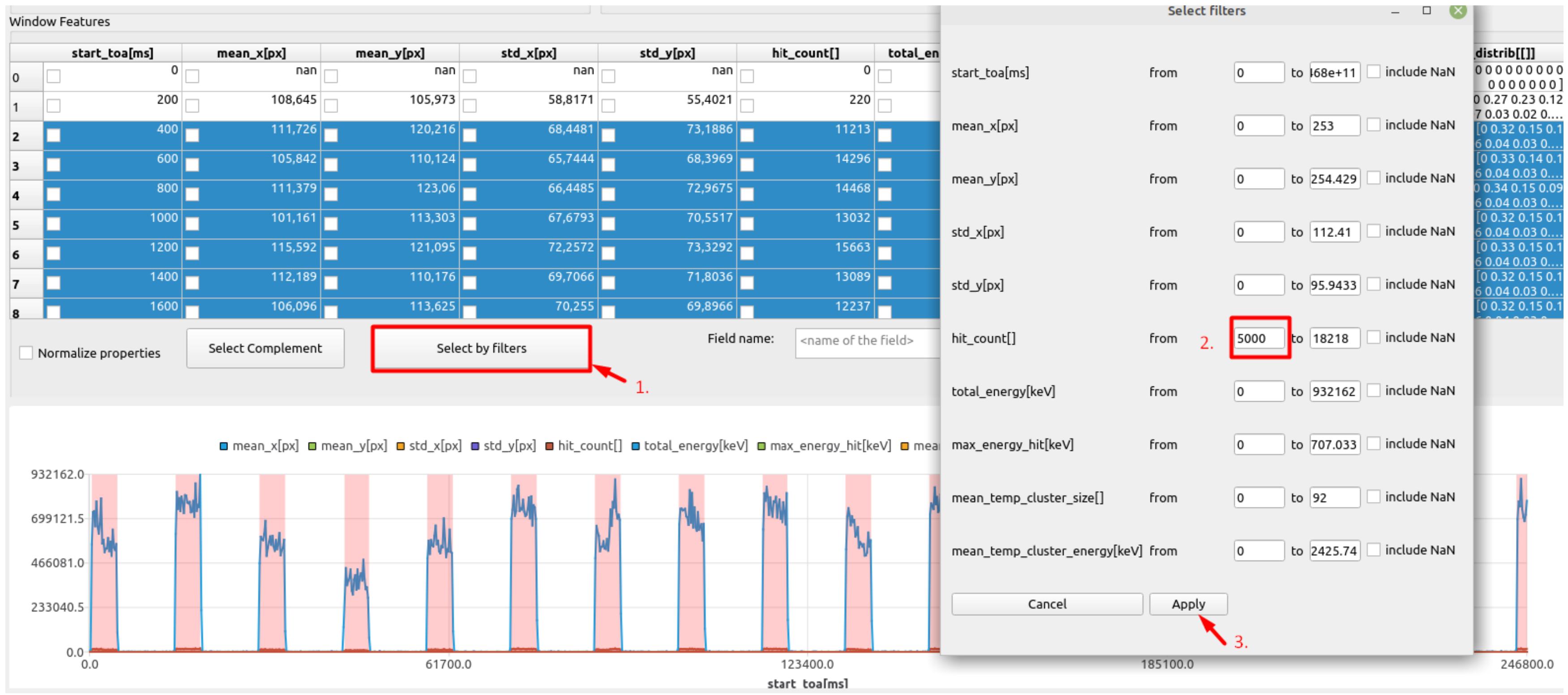
(b) Plot of differentiated feature (hit count) with respect to time.

Figure A.2: Comparison of the standard (not differentiated) features and their differentiated version.

7. Select windows of interest by hand by clicking on the leftmost field in the row (part of the table header). Keys like ‘Ctrl’ (to extend the selection) or ‘Shift’ (for interval selection) can be used in the table to select multiple windows.

8. Click ‘Select by filters’ and choose the interval for each property. Click ‘Apply’. Only windows with properties within these intervals are **added to the selection (currently selected rows are NOT unselected)**. This allows you to use this button multiple times to combine multiple interval criteria by disjunction.

9. The selected windows are highlighted in the plot below.



Window Features

| | start_toa[ms] | mean_x[px] | mean_y[px] | std_x[px] | std_y[px] | hit_count[] |
|---|---|---|---|---|---|---|
| 0 | 0 | nan | nan | nan | nan | 0 |
| 1 | 200 | 108,645 | 105,973 | 58,8171 | 55,4021 | 220 |
| 2 | 400 | 111,726 | 120,216 | 68,4481 | 73,1886 | 11213 |
| 3 | 600 | 105,842 | 110,124 | 65,7444 | 68,3969 | 14296 |
| 4 | 800 | 111,379 | 123,06 | 66,4485 | 72,9675 | 14468 |
| 5 | 1000 | 101,161 | 113,303 | 67,6793 | 70,5517 | 13032 |
| 6 | 1200 | 115,592 | 121,095 | 72,2572 | 73,3292 | 15663 |
| 7 | 1400 | 112,189 | 110,176 | 69,7066 | 71,8036 | 13089 |
| 8 | 1600 | 106,096 | 113,625 | 70,255 | 69,8966 | 12237 |

Select filters

| | | | | | |
|---|---|---|---|---|---|
| start_toa[ms] | from | 0 | to | 68e+11 | include NaN |
| mean_x[px] | from | 0 | to | 253 | include NaN |
| mean_y[px] | from | 0 | to | 254.429 | include NaN |
| std_x[px] | from | 0 | to | 112.41 | include NaN |
| std_y[px] | from | 0 | to | 95.9433 | include NaN |
| hit_count[] | from | 5000 | to | 18218 | include NaN |
| total_energy[keV] | from | 0 | to | 932162 | include NaN |
| max_energy_hit[keV] | from | 0 | to | 707.033 | include NaN |
| mean_temp_cluster_size[] | from | 0 | to | 92 | include NaN |
| mean_temp_cluster_energy[keV] | from | 0 | to | 2425.74 | include NaN |

Figure A.3: Window selection, as described by the fields above. The boxes highlighted in red correspond to the steps described in the text.

10. Type in the name for the currently selected rows.
11. Click 'Save selected' to save the selected windows to the file.
12. If you want to use supervised learning for the trigger, return to the step with window selection. Select different windows (or possibly compute features of a different burda-file), select the class name for the selected windows, and **save it to the same file**. It will NOT completely rewrite the data; it will only **append** your selection to the file.
13. If an unsupervised learning method is to be used, all of the windows can be selected.
14. If a non-ML method is to be used, you still need to create an arbitrary data file selection (the selection can be empty).
15. After multiple iterations, the trigger data file is created and can be used by the 'window trigger creator' program.

### A.1.3 Window trigger creator

The window creator app serves as a tool to create trigger files, further used by the clusterer program. On input, it expects a file with window features created by the window computer.

#### Installation

As the program itself has many dependencies from large packages, we decided against building the file into a large executable. Instead, we provide a simple alternative for installation.

Prerequisites:

- Python3 is installed and added to the system path.
- Python package manager 'pip' or 'pip3' is installed and added to the system path.

1. Clone the repository at `https://github.com/TomSpeedy/window_trigge r_creator` using HTTPS or SSH.
2. The repository contains 'requirements.txt' file where required dependencies are listed. For installation of dependencies, navigate to the 'window_trigger_creator' directory and run in terminal **'pip install -r requirements.txt'**.

3. Check the command-line output and verify that the requirements were successfully installed. In case of an error, you can contact the author.
4. Navigate to the ‘run_script’ directory.
5. For Windows OS, navigate to the ‘windows’ directory. For Linux, navigate to the ‘linux’ directory.
6. To check the installation, double click on the ‘window_trigger_creator.bat’ or ‘window_trigger_creator.sh’ – the GUI should appear.

**Usage**

1. Load the file with window feature values (*.wf). You can again type it by hand, use the ‘Browse’ button or ‘Drag and drop’ the file into the text field. Then, click ‘Load selected file’, see Figure A.4.

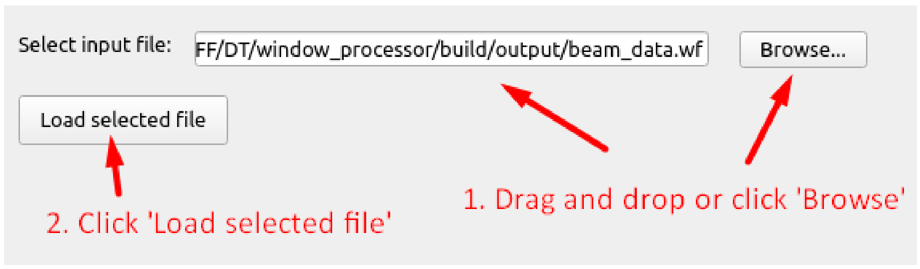
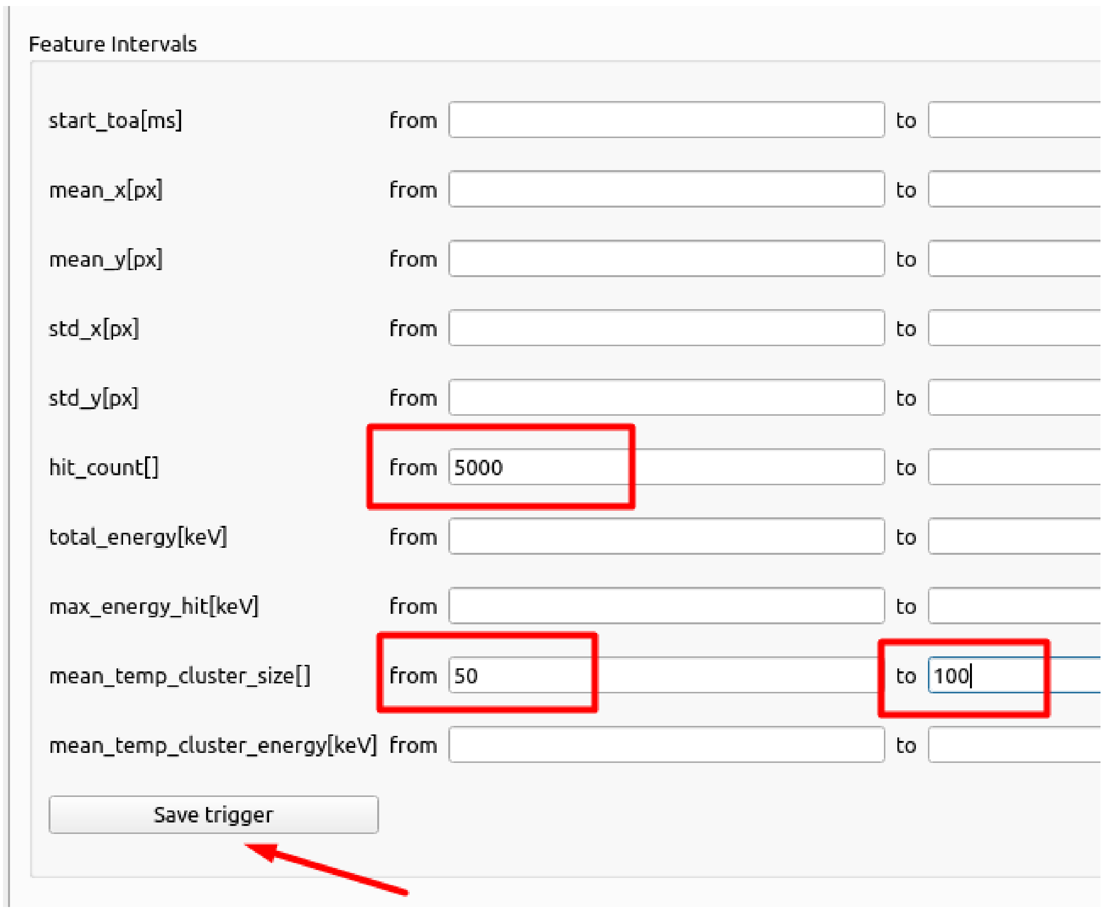

Select input file:
FF/DT/window_processor/build/output/beam_data.wf
Browse...
Load selected file
1. Drag and drop or click 'Browse'
2. Click 'Load selected file'

Figure A.4: Loading the window feature file.

2. After the file is loaded, choose the trigger type.
   - Manual interval trigger – choose the intervals of interest and click ‘Save trigger’. To combine multiple triggers by disjunction, repeat this process and save the triggers to the same file see Figure A.5. The file will NOT be overwritten, it will only append the interval selection to the file.



Feature Intervals

| | | | | |
|---|---|---|---|---|
| start_toa[ms] | from | | to | |
| mean_x[px] | from | | to | |
| mean_y[px] | from | | to | |
| std_x[px] | from | | to | |
| std_y[px] | from | | to | |
| hit_count[] | from | 5000 | to | |
| total_energy[keV] | from | | to | |
| max_energy_hit[keV] | from | | to | |
| mean_temp_cluster_size[] | from | 50 | to | 100 |
| mean_temp_cluster_energy[keV] | from | | to | |

Figure A.5: Creating the interval trigger.

- Supervised machine-learned trigger (SVM, NN), see Figure A.6.
  (a) Optionally, import hyperparameter file (again, you can use the 'Browse' button or 'Drag and drop'). The file is in JSON format, and the parameters are listed in Table A.1 and A.2.
  (b) Choose the classes which should trigger the clustering.
  (c) Click 'Start Training' and wait until the training is done. (It is shown in the log on the right.)
  (d) Click 'Save trained model'.

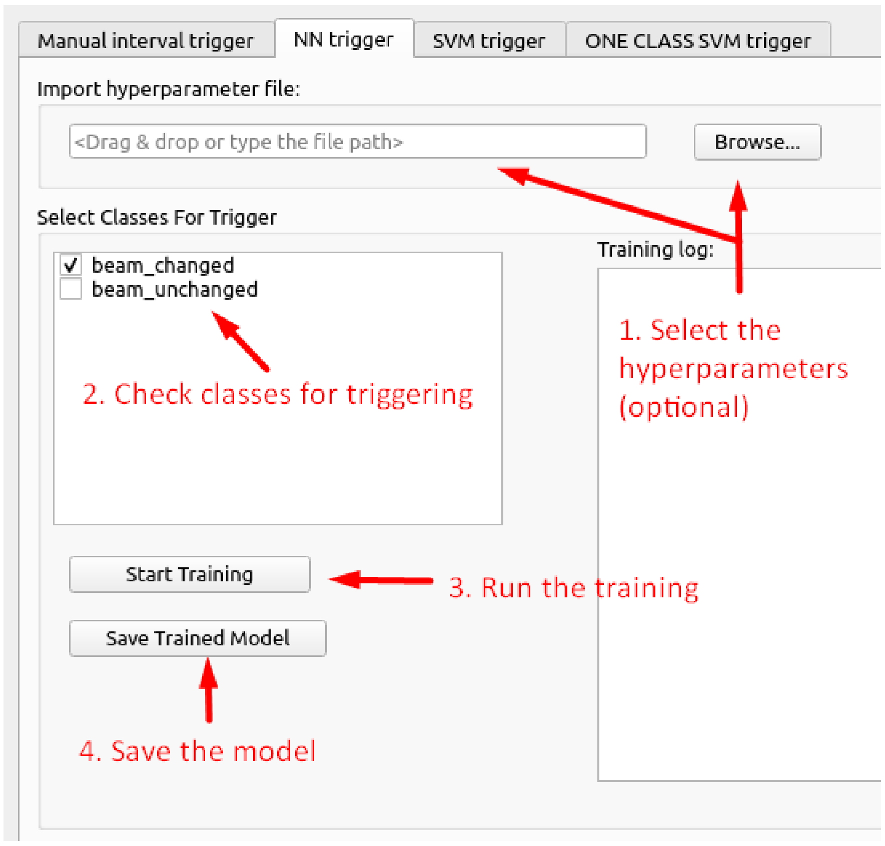


Figure A.6: Training supervised trigger.

- Unsupervised machine-learned trigger (one class SVM), see Figure A.7.
  - (a) Optionally, import hyperparameter file (again, you can use the 'Browse' button or 'Drag and drop'). The file is in JSON format, and the parameters are the same as in Table A.2, **except for the 'dataSplit' parameter**, which is left out (no splitting is taking place).
  - (b) Click 'Start Training' and wait until the training is done. (It is shown in the log on the right.)
  - (c) Click 'Save trained model'.

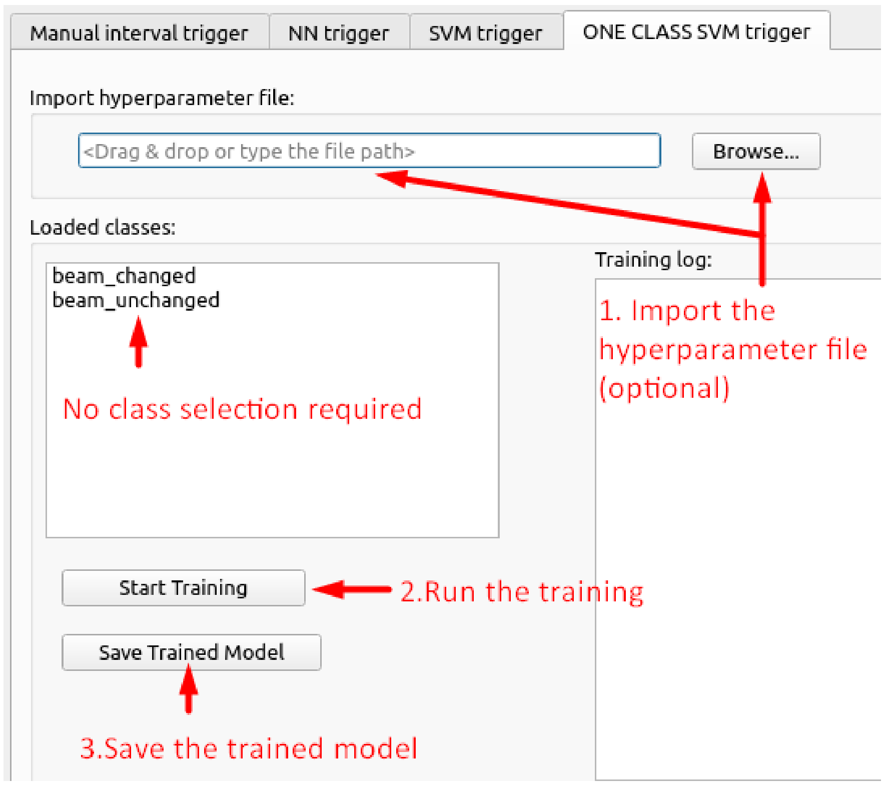


Figure A.7: Training an unsupervised trigger.

After the models are saved, they can be passed to the clusterer within the 'node args file' ('--args' option) as an argument 'trigger_file' to the 'trigger' node.

| MLP hyperparameters | | |
|---|---|---|
| Name | Type | Additional constraints |
| learningRate | floating point | It has positive value, recommended between $10^{-6}$ and $10^{-2}$. The default is $10^{-3}$ |
| layerSizes | list of integers | It is a non-empty list of positive integers. The default is $[20, 20]$ |
| learningRateDecay | dictionary | It contains fields 'name', 'initialLearningRate', and if 'name' is 'exponential' also 'decayRate'. It is optional, it has no default (no decay). |
| learningRateDecay .name | string | Is one of 'exponential' or 'cosine'. |
| learningRateDecay .initialLearningRate | floating point | It has positive value, recommended between $10^{-6}$ and $10^{-2}$. |
| learningRateDecay .decayRate | floating point | It has positive value smaller than 1. |
| optimizer | string | Currently only 'Adam' is supported. |
| batchSize | integer | It has a positive value, the default is 5. |
| epochCount | integer | It has a positive value, the default is 30. |
| dataSplit | dictionary | It contains three fields 'training' 'test' and 'validation', each floating point number between 0 and 1, summing to 1. The default is 0.7 for training, 0.15 for validation, and 0.15 for testing. |

Table A.1: Allowed hyperparameters of the multilayered perceptron and their domains.

| SVM hyperparameters | | |
|---|---|---|
| Name | Type | Additional constraints |
| kernel | string | It is one of 'rbf', 'linear', 'poly', 'sigmoid'. The default is 'rbf'. |
| gamma | string | It is one of 'scale' or 'auto'. The default is 'auto'. |
| dataSplit | dictionary | It contains three fields, 'training' and 'test', each floating point number between 0 and 1, summing to 1. The default is 0.8 for training and 0.2 for testing. |

Table A.2: Allowed hyperparameters of the support vector machines and their domains. For more information, see `https://scikit-learn.org/stable/modules/svm.html`.

### The ONNX file format and extensibility

The design of the software was made in such a way that it is possible to extend the list of supported ML methods without breaking changes to the source code. This enables us to try different triggers in the future and compare their effectiveness. The key to the extensibility of the program is the uniform model format. After the training, each model is converted to an Open neural network exchange format. The ONNX format provides a uniform interface to heterogeneous models. Additionally, it serves as a bridge between the model's training (typically performed in Python programming language) and the performance-critical inference of the model (done in C++ language). In principle, it is possible to bypass the window trigger creator program and use an arbitrary ML model with custom training. However, it is very important to preserve the same input and output signature of the model and convert it to the ONNX file format. Nevertheless, it is best to consult such customization with the author.

### Contact

Should you have ideas for improvement, problems, or questions related to the software, don't hesitate to get in touch with us at celko.tom@gmail.com. We hope that the developed tools will help you in your work.