Fachbereich Informatik und Mathematik
Institut für Informatik

Lehrstuhl für Architektur von Hochleistungsrechnern

Bachelorarbeit

Kilian Hunold

Thema:	Optimierung der lokalen Rekonstruktion im Micro Vertex Detector des CBM-Experiments
Matrikelnr.:	6045577
Version vom:	17. April 2019
Betreuer:	Prof. Dr. Volker Lindenstruth

Erklärung gemäß Bachelor-Ordnung Informatik 2011§25 Abs. 11

Hiermit erkläre ich, dass ich die Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe.

Ort, den Datum

Vorname Nachname

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die hier vorliegende Arbeit ist Teil des CBM (Compressed Baryonic Matter) Experiments, welches an der GSI Helmholtzzentrum für Schwerionenforschung durchgeführt werden soll (vergleiche Facility for Antiproton and Ion Research, 2019 [4]). CBM selbst ist Bestandteil des im Bau befindlichen FAIR (Facility for Antiproton and Ion Research). Das CBM Experiment (siehe P. Senger and V. Friese, 2012 [10]) befasst sich mit stark verdichteter nuklearer Materie. Es werden hierbei unterschiedliche Stadien und Phasen von stark interagierender Materie betrachtet. Im Genauen wird das Quark-Gluon-Plasma erforscht, welches ein besonderer Aggregatzustand ist. Um diesen zu erreichen wird eine Kollision von Schwerionen erzeugt, sodass sich die Quark-Struktur im Inneren des Nukleons löst, diese sich frei bewegen können und somit das Quark-Gluon-Plasma entsteht. Diese Forschung wird betrieben um beispielsweise Rückschlüsse auf das Verhalten von Neutronensternen zu ziehen, um herauszufinden, warum es keine isolierten Quarks gibt und wie die Masse von Nukleonen entsteht. Hierbei wird beispielsweise die Anzahl, die Masse, die Energie und die Zusammensetzung der Teilchen gemessen, um Rückschlüsse auf Dichte oder Temperatur des dort entstandenen Feuerballs zu ziehen.

Im CBM Experiment sind nicht alle Ereignisse gleich häufig und einige Teilchen sind hierbei nur für kurze Zeit vor ihrem Zerfall zu erkennen. Daher werden im CBM-Experiment möglichst viele Kollisionen in kürzester Zeit erzeugt, sodass diese Ereignisse häufiger eintreffen und so erforscht werden können. Es wird davon ausgegangen, dass ca. zehnmillionen Kollisionen pro Sekunde erzeugt und untersucht werden können. Dieses Experiment soll in einigen Fällen, wie zum Beispiel bei Elektron-Positron-Paaren, Myonenpaaren oder schweren Quarks zum ersten Mal Messungen anstellen. Des Weiteren soll das Ausbreitungsverhalten der einzelnen Teilchen erforscht werden, um unter anderem Rückschlüsse auf den Urknall zu ziehen.
Um hier aussagekräftige Forschung zu leisten, werden sehr leistungsfähige und schnelle Detektoren benötigt. Daher existieren sechs Detektoren, die durch unterschiedlichste Verfahren die unterschiedlichen Eigenschaften der Teilchen erforschen sollen. Maximal arbeiten diese mit einer Datenrate von etwa 10 Mhz.
Da die Detektoren pro Sekunde ca. einen Terabyte an Daten liefern, werden Hochleistungsrechner benötigt um diese Daten live zu verarbeiten und anschließend auswerten zu können. Hiefür ist an der GSI der sogenannte

FLES (First Level Event Selector siehe De Cuveland, J and Lindenstruth, V and CBM collaboration and others, 2011 [2]) zuständig. Dieser übernimmt die Daten, welche die Detektoren ihm liefern und führt bereits erste Bearbeitungsschritte durch. Er ist im Allgemeinen dafür zuständig eine Track-Reconstruction in 4D (3D + Zeit) zu erzeugen, zeitbasierte Events zu erstellen und seltene und besondere Ereignisse zu filtern. Außerdem wird er dazu genutzt das miniCBM (eine kleine Version des CBM-Experiments zu Testzwecken) zu überprüfen und bei der Entwicklung Fehler zu überprüfen. Um den Datenstrom zu bewältigen wird der FLES mit ca. 1000 Inputnodes ausgestattet. Die Onlineauswertung wird auf ca. 60.000 Prozessorkernen betrieben. Die Inputnodes verteilen hierbei die erhaltenen Daten der Detektoren auf unterschiedliche Processnodes, sodass Daten mit gleichem Timestamp auf dem gleichen Processnode verarbeitet werden. Daraus resultierend hat jeder Processnode einen anderen Timestamp, den er verarbeitet. Der hier behandelte Programmabschnitt könnte auf den Processnodes genutzt werden, um eine schnelle Verarbeitung der Daten des MVD zu gewährleisten.

1.2 Struktur der Arbeit

Abschnitt 1 beschreibt in der Motivation grob das CBM-Experiment. In Kapitel 2 werden die Grundlagen der Arbeit geschaffen. Es werden notwendige Begrifflichkeiten erläutert und die Grundlagen zum Verständnis dieser Arbeit gesetzt.
Anschließend werden in Kapitel 3 die Ausgangs-Algorithmen von Cluster- & Hitfinder explizit vorgestellt und erklärt, sodass in Kapitel 4 die Optimierungen im Vordergrund stehen.

In Kapitel 4 werden die behandelten Optimierungen vorgestellt. Es wird hier erläutert, welche Optimierungen am betrachteten Code vorgenommen wurden und warum diese relevant sind im Bezug auf die Performanz.
Zuerst werden die Optimierungen des Clusterfinders behandelt. Es wird mit der Variablenoptimierung begonnen. Anschließend werden Optimierungen der Datenstrukturen und ein neuer Algorithmus betrachtet. Hierbei wird ebenfalls ein parallelisierter Algorithmus in Betracht gezogen.
Im zweiten Teil des Kapitels wird der Hitfinder in den Programmabschnitt des Clusterfinders integriert und somit die ursprünglichen Tasks Cluster- und Hitfinder durch den neuen Task namens "DigiToHit" ersetzt.
Zuletzt folgt eine Zusammenfassung der Ergebnisse, die im Rahmen dieser Arbeit erbracht wurden.

2 Grundlagen und Voraussetzungen

2.1 Allgemeine Informationen zur Simulation und Ausführung

Die hierbei vorliegenden Eingabedaten wurden ausschließlich simuliert und anschließend digitalisiert, da sich sowohl die Beschleuniger als auch die Detektoren noch im Bau befinden.
Die Performanz des Programmes wurde sowohl durch eine Eingabe von Daten resultierend aus simulierten Minimum-Bias-Kollisionen als auch aus zentralisierten Kollisionen gemessen, bei welchen geladene Goldionen auf ein Goldtarget treffen.
Die Minimum-Bias-Kollision beschreibt den realen Fall, dass die Teilchen nicht zentral aufeinander treffen. Hier entgegen steht die Simulation der zentralen Kollision. Diese ist idealisiert und betrachtet nur die Teilchen, die exakt aufeinander getroffen sind.
Der MVD befasst sich derzeit nur mit der Event-basierten Art der Datenverarbeitung. Somit wurde die zeitbasierte Verarbeitung in dieser Arbeit nicht behandelt.
Bei dem sich in der Planung befindenden Experiment wird später der Sis100 Teilchenbeschleunigers genutzt. Dieser wurde auch in die Simulationsschritte mit einbezogen, um möglichst ähnliche Daten wie bei dem Originalexperiment zu erhalten. Für die Datenprozessierung wurde der miniFLES gebaut, welcher eine kleine Version des später verwendeten FLES darstellt. Dieser kommt zum Einsatz, um sowohl Soft als auch Hardware für den in Planung stehenden FLES zu testen. Somit wird die Performanz zu zukünftigen Ergebnissen abweichen. Zur Betrachtung der Geschwindigkeit und Performanz-Steigerung der Tasks ist es aber dennoch aussagekräftig.

2.2 Performanzmessung aller Detektoren

Um herauszufinden welcher der Detektoren den höchsten Optimierungsbedarf aufweist, wurde zu Anfang der Zeitverbrauch aller lokaler Rekonstruktionen der Detektoren des CBM Experiments gemessen. In Abbildung 1 und Abbildung 2 ist gezeigt, wie viel Zeit jeder Detektor pro Event benötigt.
Die dunkelblauen Balken zeigen das Maximum der im Event verbrauchten Zeit. Die Mittleren zeigen das Minimum an und die grünen Balken beschreiben den berechneten Mittelwert der Zeiten pro Event der unterschiedlichen Detektoren. Es ist hier zu sehen, dass der MVD (Micro Vertex Detector) sowohl im Maximum als auch im Minimum verglichen mit den anderen De-

tektoren mehr Zeit benötigt. Somit wurde entschieden sich in dieser Arbeit mit der Performanz des Micro Vertex Detectors auseinanderzusetzen.
Die hier vorliegende Messung wurde im Gegensatz zu den anderen Messungen mit den Daten einer simulierten zentralen Kollision erzeugt.
(Tasks von Links nach Rechts in Abbildung 1 und Abbildung 2:
FairTaskList, MVDClusterFinder, MVDHitFinder, StsFindClusters, StsFindHits, TrdClusterFinder, TrdHitProducer, TOFSimpleClusterizer, IdealPsdHitProducer)

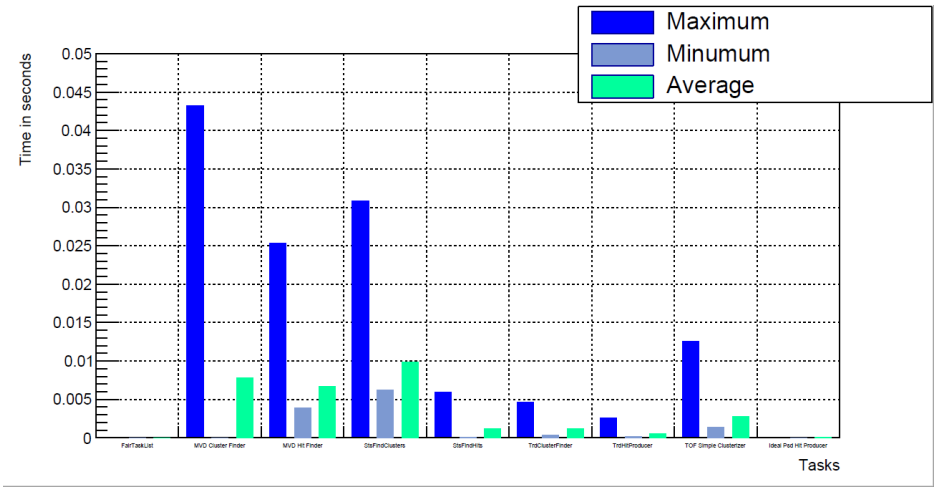

Abbildung 1: Zeit pro Event aller Detektoren mit Minimumbias Events

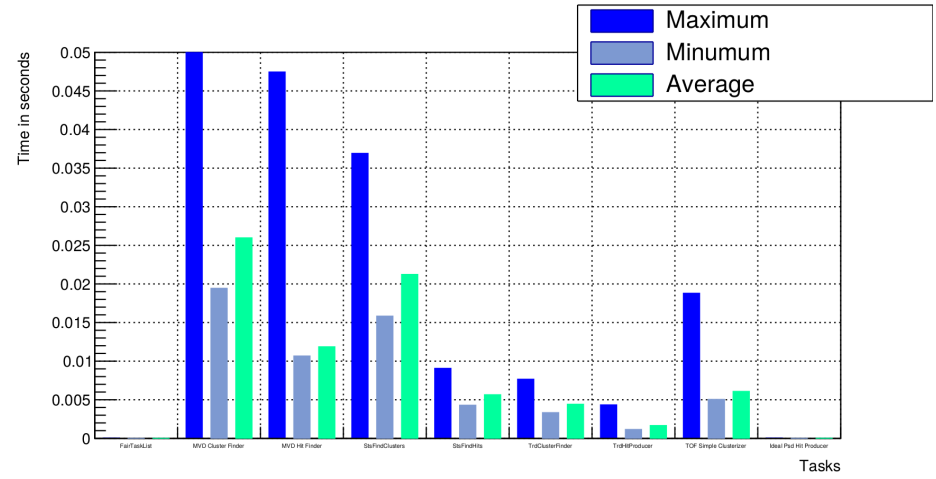

Abbildung 2: Zeit pro Event aller Detektoren mit zentralisierten Events

2.3 Der Micro Vertex Detector

Der MVD(siehe Deveaux,2009 [3]) ist der erste in der Reihe der Detektoren und steht somit am nächsten zum Target. Dieser ist mit einer hohen Granularität ausgestattet, um eine möglichst genaue Auflösung der Teilchen zu erhalten. Der Detektor verfügt derzeit über ca. 300 CMOS Sensoren, welche die Ladungen und Positionen der darauf treffenden Teilchen ermitteln. Der Detektor ist in vier Stationen aufgeteilt. Diese stehen jeweils im Abstand von 50 mm hinter dem Target. Jede Station ist mit einer unterschiedlichen Anzahl von Sensoren ausgestattet und hat darauf basierend unterschiedliche Ausmaße. Die erste Station hat acht, die zweite 40, die dritte 84 und die letzte 160 Sensoren. Die vier Stationen sind in Abbildung 3, einer schematischen Darstellung des Micro Vertex Detectors, eindeutig nachzuvollziehen.
Die Vorstellung des Detektors innerhalb der Simulation hingegen orientiert sich an den Sensoren. Wir betrachten die entgegengenommenen Daten wie eine zweidimensionale Matrix, welche eine Höhe von 576 und eine Breite von 1152 Pixeln hat(ähnlich Abbildung 4). Somit wird jeder Sensor einzeln behandelt und ausgewertet. Man kann daher im Nachhinein die Abarbeitung der einzelnen Sensoren parallelisieren, sodass die Sensoren nicht nacheinander abgearbeitet werden.

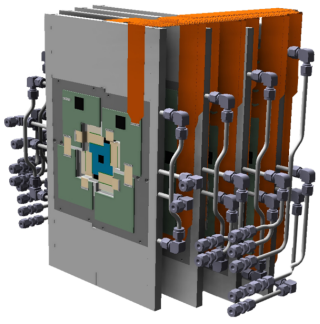

Abbildung 3: MVD

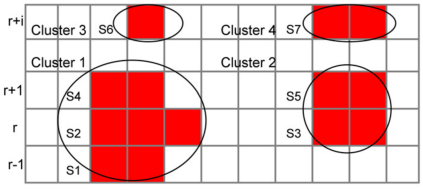

Abbildung 4: Sensordarstellung

Der MVD erforscht verschiedene Teilchenbahnen, indem direkt hinter dem Target innerhalb eines Magnetfeldes mit der hohen Auflösung des MVD gemessen wird.

2.4 CbmRoot/FairRoot

CbmRoot (vergleiche GSI, 2018 [6]) und FairRoot (vergleiche GSI, 2018 [7]) sind Frameworks welche im Bereich des Cbm Eperiments am FAIR genutzt werden. Sie beinhalten Funktionen und Datenstrukturen, welche optimal auf das Experiment angepasst sind. Sie dienen zur Simulation, Rekonstruktion und Analyse von physikalischen Daten. Sie sind darauf ausgelegt physikalisch-technische Daten schnell und effizient zu verarbeiten. Das Framework beinhaltet Schnittstellen, welche genutzt werden um die generierten und gemessenen Daten zwischen den einzelnen Auswertungsschritten zu transferieren. So wird hier beispielsweise die Struktur eines Digis oder eines Clusters definiert, sodass sie anschließend in dem hier behandelten Programmabschnitt genutzt werden können, um die Daten zwischen den Tasks zu übergeben.
Des Weiteren beinhaltet das Framework Funktionen, welche es erlauben Histogramme zu erstellen und sie in einer von Root vorgegebenen Struktur auszugeben, sodass diese nicht nur angezeigt werden sondern die Daten zur Erstellung des Histogramms im Anschluss zur weiteren Verarbeitung ausgelesen werden können.
Das Root-Framework wurde zur Erstellung aller Histogramme und Zeitmessungen verwendet, welche in dieser Arbeit abgebildet wurden.

2.5 Wörterbuch

In der vorliegenden Arbeit werden einige Begrifflichkeiten häufiger verwendet. Um hier vorab die Grundlagen zu schaffen und Missverständnisse auszuschließen, werden im nächsten Abschnitt diese Begrifflichkeiten erläutert.

1. Hit:
   Ein Hit ist der zentrale Punkt eines Clusters. Dieser wird im Laufe dieser Arbeit unter anderem berechnet. Er beschreibt die Stelle, an welcher ein Teilchen die größte Wechselwirkung mit dem Detektor innerhalb eines Clusters hat.

2. Cluster/CbmMvdCluster:
   Ein Cluster beschreibt einen Zusammenschluss von einem oder mehreren Hits. Diese müssen auf den Detektor-Platten direkt nebeneinander liegen.
   Der CbmMvdCluster ist die dazugehörige Datenstruktur und besteht aus folgenden Variablen:

Typ	Name	Beschreibung
std::map<std::pair <Int_t, Int_t >, Int_t>	fPixelMap	Key-Value-Paare aus Koordinaten und Ladung.
Int_t	fRefId	Identifikationsnummer eines Clusters
Float_t	fClusterCharge	Ladung des Clusters

3. Digi/CbmMvdDigi:
   Ein Digi beschreibt die Wechselwirkung eines Teilchens mit der Detektoroberfläche. Diese werden anschließend digitalisiert und in einem Array abgespeichert. Ein Digi ist somit die elektronische Antwort des Detektors. Diese wird in der dazugehörigen Datenstuktur CbmMvdDigi abgelegt, um sie für folgende Verarbeitungsschritte strukturiert weitergeben zu können. Da in dem betrachteten Programmabschnitt nicht alle Informationen über einen Digi benötigt werden, folgt nun eine gekürzte Auflistung der Datenstruktur:

Typ	Name	Beschreibung
Int_t	iChannelNrX	X-Koordinate
Int_t	iChannelNrY	Y-Koordinate
Float_t	charge	Ladung des Hits
Float_t	time = 0.0	Zeitpunkt der Messung

4. Charge:
   Die Charge beschreibt die Ladung eines Teilchens, welche an dem Ort der Kollision auf der Detektoroberfläche gemessen wurde.

5. AdcCharge
   Die AdcCharge ist eine Weiterverarbeitung der gemessenen Ladung eines Digis. Im Abschnitt 3.1.2 (Hilfsfunktion GetAdcCharge) wird diese Funktion genauer erläutert.

6. Inputbuffer/Outputbuffer
   Bei diesen beiden Buffern handelt es sich um TClonesArray welche in dem RootFramework definiert sind. Sie dienen als Übergangsdatenstruktur zwischen unterschiedlichen Tasks und speichern beispielsweise Digis, Cluster oder Hits.

7. TClonesArray
   Ein TClonesArray ist eine vom Root-Framework gelieferte Datenstruktur, welche es ermöglicht viele Daten schnell speichern und lesen zu können. Es ist eine Liste an gespeicherten Elementen und wird im folgendem Programmkontext als Input- und Outputdatenstruktur verwendet, um Ergebnisse zwischen den einzelnen Tasks zu übergeben.

3 Ausgangsalgorithmen

3.1 Clusterfinder

Der Clusterfinder ist eine lokale Struktur des Micro Vertex Detectors und verbraucht hierbei den größten Zeitanteil. Bei diesem geht es darum, Cluster aus einer Eingabe von Digis herauszusuchen und an den Hitfinder, die zweite der hier bearbeiteten Strukturen weiterzugeben. Die Eingabe an den Clusterfinder ist eine TClonesArray (siehe 2.5) von Digis. In dieser sind viele Informationen über den Digi selbst abgespeichert, wovon schlussendlich in dieser Arbeit nur die Koordinaten und die jeweilige Ladung gebraucht werden.
Es wird hier nur der hauptsächliche Algorithmus betrachtet. Schritte, welche abseits des Algorithmus passieren, werden bei den Optimierungen erwähnt.

3.1.1 Anfänglich initialisierte Variablen des Clusterfinders

Zu Anfang werden zwei TClonesArrays initialisiert, wobei das eine die Input- und das andere die Outputdatenstruktur darstellt. Beide werden mit einer Größe von 10000 CbmMvdDigis ausgestattet.
Der Inputbuffer enthält CbmMvdDigis. Diese bestehen, wie bereits im Abschnitt 2.5 beschrieben, aus mehreren Variablen. Im Clusterfinder werden davon maßgeblich die Koordinaten des Digis und die jeweilige Ladung benötigt.
Der Outputbuffer hingegen besteht aus CbmMvdClustern. Als nächstes wird das PixelUsed Array mit der Größe von dem InputBuffer erzeugt und alle Werte innerhalb dieses Arrays auf den Integer Null gesetzt.
Nun wird die DigiMap erzeugt, welche alle Digis des Inputbuffers als Key-Value-Paar speichert. Der Schlüssel sind hierbei die Koordinaten und Verweisen auf die Ladung des Digis. Die Struktur ist in diesem Fall parallel um Inputbuffer aufgebaut, sodass die Indices beider Arrays

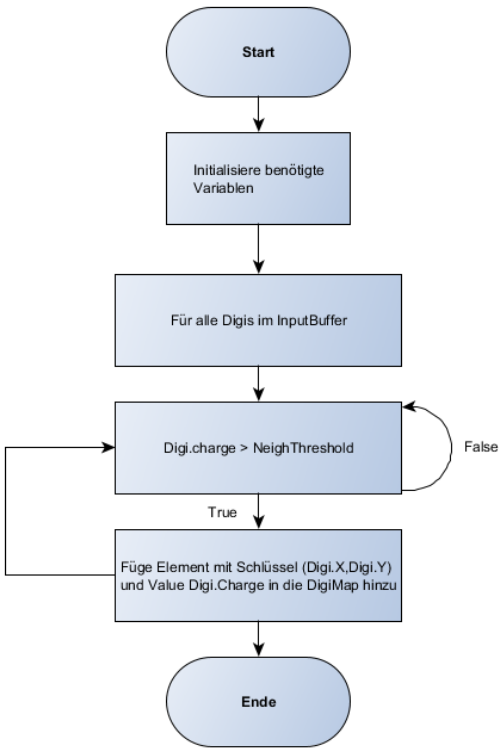

Abbildung 5: Initialisierung der DigiMap

auf den gleichen Digi verweisen. Ein Digi wird hierbei aber nur in die DigiMap übernommen, wenn er den Schwellenwert NeighThreshold (anfänglich gleich "1") überschreitet.
Die Initialisierung wird schematisch in Abbildung 5 dargestellt.

3.1.2 Hilfsfunktion GetAdcCharge

"GetAdcCharge" (schematisch dargestellt in Abbildung 6) ist eine Funktion für die Weiterverarbeitung der Ladung eines Digis im Bezug auf einen Offset und eine Schrittgröße. Hierbei wird überprüft, ob die Ladung kleiner als der gegebene Offset ist. Tritt dieser Fall ein, so ist die daraus resultierende AdcCharge gleich null. Andernfalls wird überprüft, ob die auf die Ganzzahl gerundete Division von Charge minus Offset und der Schrittgröße größer ist als AdcSteps minus 1. Bewahrheitet sich die Abfrage, so ist die AdcCharge gleich der AdcSteps - 1. Tritt dieser Fall nicht ein, so ist die AdcCharge die zuvor berechnete Division.

3.1.3 Hilfsfunktion Find Neighbours

"Find Neighbours" (schematisch dargestellt in Abbildung 7) ist eine der zentralen Funktionen des Clusterfinders, welche die direkten Nachbarn eines gegebenen Digis sucht und speichert. Diese wird im Kontext des Ausgangsalgorithmus in einer Art rekursiven Schleife ausgeführt, um alle aneinanderliegenden Digis und somit einen Cluster zu finden.
Die Funktion bekommt drei Argumente übergeben. Zum einen das ClusterArray, welches, wie schon zuvor beschrieben, alle Indices der bereits gefundenen Elemente eines Clusters speichert. Des Weiteren erhält diese Funktion einen ClusterDigi, welches der Index des zurzeit betrachteten Digis ist, zu welchem die Nachbarn gefunden werden sollen. Der dritte Parameter ist das pixelUsed Array, welches die bereits betrachteten Pixel beinhaltet.
Die Funktion holt sich zunächst die Koordinaten des zurzeit betrachteten Digis aus dem InputBuffer. Diese werden für jede Richtung einmal in- und dekrementiert ((X+1,Y);(X-1,Y);(X,Y+1);(X,Y-1)). In der unten stehenden Abbildung 7 wird dies nur für eine Richtung exemplarisch durchgeführt.
Anschließend kontrolliert der Algorithmus, ob der nebenstehende Pixel (z.B. (X+1,Y)) in der DigiMap vertreten ist. Tritt dieser Fall ein, so wird der Index des Digis im ClusterArray gespeichert, dieser im PixelUsed Array als genutzt markiert und anschließend aus der DigiMap gelöscht.
Wird der Pixel nicht in der DigiMap gefunden, so endet die Funktion an dieser Stelle.

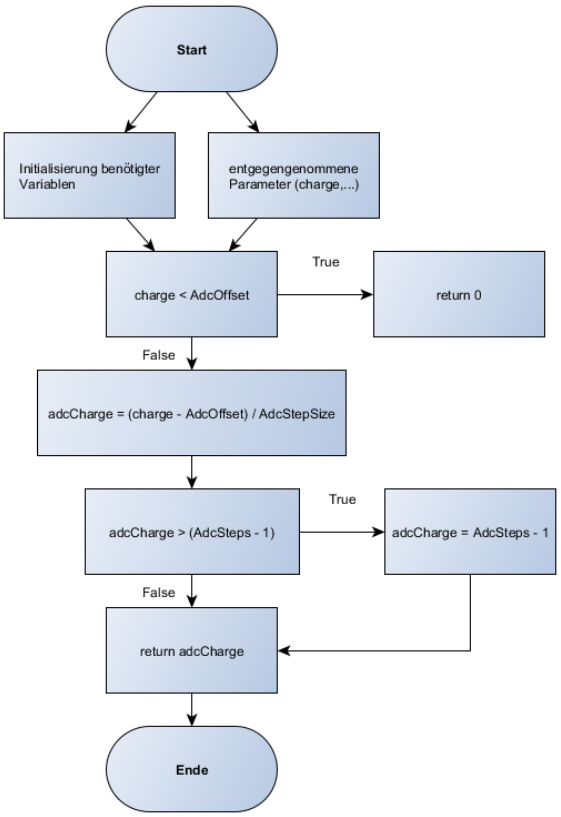

Abbildung 6: Get_Adc_Charge

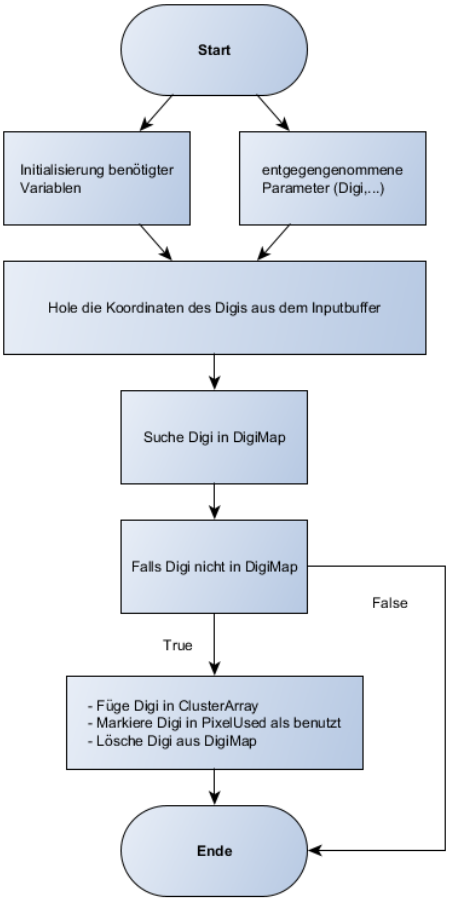

Abbildung 7: Find_Neighbours

3.1.4 Der Ausgangsalgorithmus des Clusterfinders

Im folgenden Abschnitt wird der Algorithmus des Clusterfinders beschrieben (schematische Darstellung in Abbildung 8). Dieser sucht nach direkt nebeneinander liegenden Digis.
Für jedes Element im InputBuffer soll nachgeschaut werden, ob die adcLadung größer gleich des gegebenen SeedThresholds ist. Dieser ist in unserem Fall auf 1 gesetzt und kann gegebenenfalls im Nachhinein noch angepasst werden. Außerdem wird kontrolliert, ob der betrachtete Digi bereits im pixelUsed Array auf Null gesetzt wurde. Ist dies der Fall, so wird das Element mit den Koordinaten von unserem betrachteten Digi aus der DigiMap gelöscht und das Element mit selbigem Index im PixelUsed Array auf ”1” gesetzt.

Zum Schluss wird der derzeitig gewählte Digi noch mit seinem Index im ClusterArray gespeichert.
Nun wird über alle Elemente des ClusterArrays iteriert.
Für jedes Element im ClusterArray wird geschaut, ob ein benachbarter Digi (wie schon in Abschnitt 3.1.3 beschrieben) in der DigiMap vertreten ist. Wird ein solcher gefunden, so fragt der Algorithmus den jeweiligen Index des gefundenen Digis ab. Nun wird der Wert an der Stelle des Index im PixelUsed Array auf "1" gesetzt und das Element aus der DigiMap entfernt. Als letztes wird der Index des Elements noch im ClusterArray festgehalten. Da wir nun ein Element mehr im ClusterArray haben, geht der Algorithmus rekursiv weiter und sucht die Digis im Umfeld des neu hinzugefügten Digis.
Im Anschluss an das Suchen des Clusters wird dieser als Output in eine neue Map geschrieben. Darauffolgend wird die Map mit den Pixeln des Clusters in den Outputbuffer übernommen, sodass der Hitfinder die Daten übernehmen und diese im Anschluss weiterverarbeiten kann.

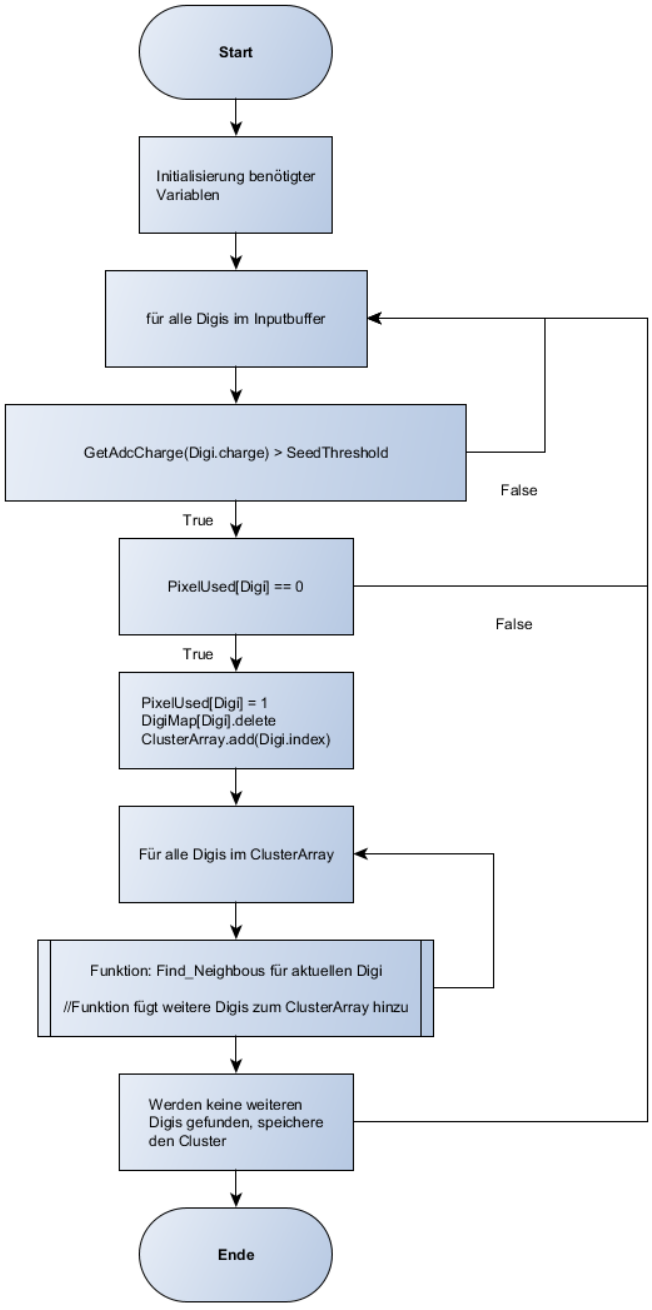

Abbildung 8: Ausgangsalgorithmus des Clusterfinders

Name	Typ	Beschreibung
numeratorX	Double_t	$\sum_{digi \in Cluster}(X_{digi} \cdot Q_{digi})$
numeratorY	Double_t	$\sum_{digi \in Cluster}(Y_{digi} \cdot Q_{digi})$
denumerator	Double_t	$\sum_{digi \in Cluster}(Q_{digi})$
x	Int_t	X-Wert des aktuell betrachteten Digis (Koordinate)
y	Int_t	Y-Wert des aktuell betrachteten Digis (Koordinate)
layerPosZ	Double_t	Z-Wert des fSensors
shape	UInt_t	Form eines Clusters

Tabelle 1: Initialisierte Variablen des Hitfinders

3.2 Hitfinder

Im nächsten Abschnitt wird der Algorithmus des Hitfinders erläutert. Hier wird das gewichtete Mittel für einen Cluster berechnet und anschließend die Daten für weitere Programmabschnitte aufbereitet.

3.2.1 Anfänglich initialisierte Variablen des Hitfinders

Anfangs werden, ähnlich wie beim Clusterfinder, zwei TClonesArrays erstellt, welche den Input- und den Outputdatensatz darstellen. Diese wurden auch hier, wie beim Clusterfinder, mit einer Größe von 10000 Elementen initialisiert. Der InputBuffer besteht hierbei aus CbmMvdClustern, den der Hitfinder von dem Clusterfinder übergeben bekommt. Der Outputbuffer hingegen wird schlussendlich CbmMvdHits enthalten.

3.2.2 Der Ausgangsalgorithmus des Hitfinders

Für jedes Element im InputBuffer, welches einen zuvor berechneten Cluster darstellt, werden die in Tabelle 1 dargestellten Variablen initialisiert. Für jeden Digi des betrachteten Clusters wird die X- und Y-Koordinate abgefragt und in einer Variable gespeichert. Falls bisher weniger als vier Digis betrachtet wurden, so wird auf den derzeitigen Shape unter folgender Rechenvorschrift $2^{(4*(yIndex-yIndex0+3))+(xIndex-xIndex0)}$ hinzu addiert. Die Shape ist eine Vorberechnung der Form des Clusters, um später bessere Aussagen über Sigma- und Shift-Werte treffen zu können.
Anschließend werden die Koordinaten durch die vom Framework bereitgestellte Funktion "PixelToTop" transformiert (PixelToTop wandelt die Werte in globale Koordinaten um), mit der jeweiligen Ladung multipliziert und das Ergebnis zum numeratorX, beziehungsweise numeratorY addiert. Ebenfalls wird die Ladung des Digis selbst zum Denominator addiert.

Somit entstehen schlussendlich drei Summen für jeden Cluster. Sollte der Denominator gleich null sein, so werden die Koordinaten des Hits auf "0" gesetzt. Anderenfalls wird der X-Wert auf das gewichtete Mittel ($\frac{numeratorX}{denominator}$), der Y-Wert auf das gewichtete Mittel ($\frac{numeratorY}{denominator}$) und die Z-Koordinate auf eine Konstante gesetzt, welche durch das Framework definiert wird (CbmMvdSensor GetZ). Als nächstes wird die zuvor berechnete Shape ausgewertet. Hat die Shape eine der gegebenen Zahlen angenommen, so werden bestimmte Sigma- und Shiftwerte zugewiesen, anderenfalls existieren default Shift- und Sigmawerte, welche angenommen werden können. Diese Werte werden nun dazu genutzt, um Anpassungen an der zuvor berechneten Position des Hits vorzunehmen und so die Position des Hit exakt zu definieren.

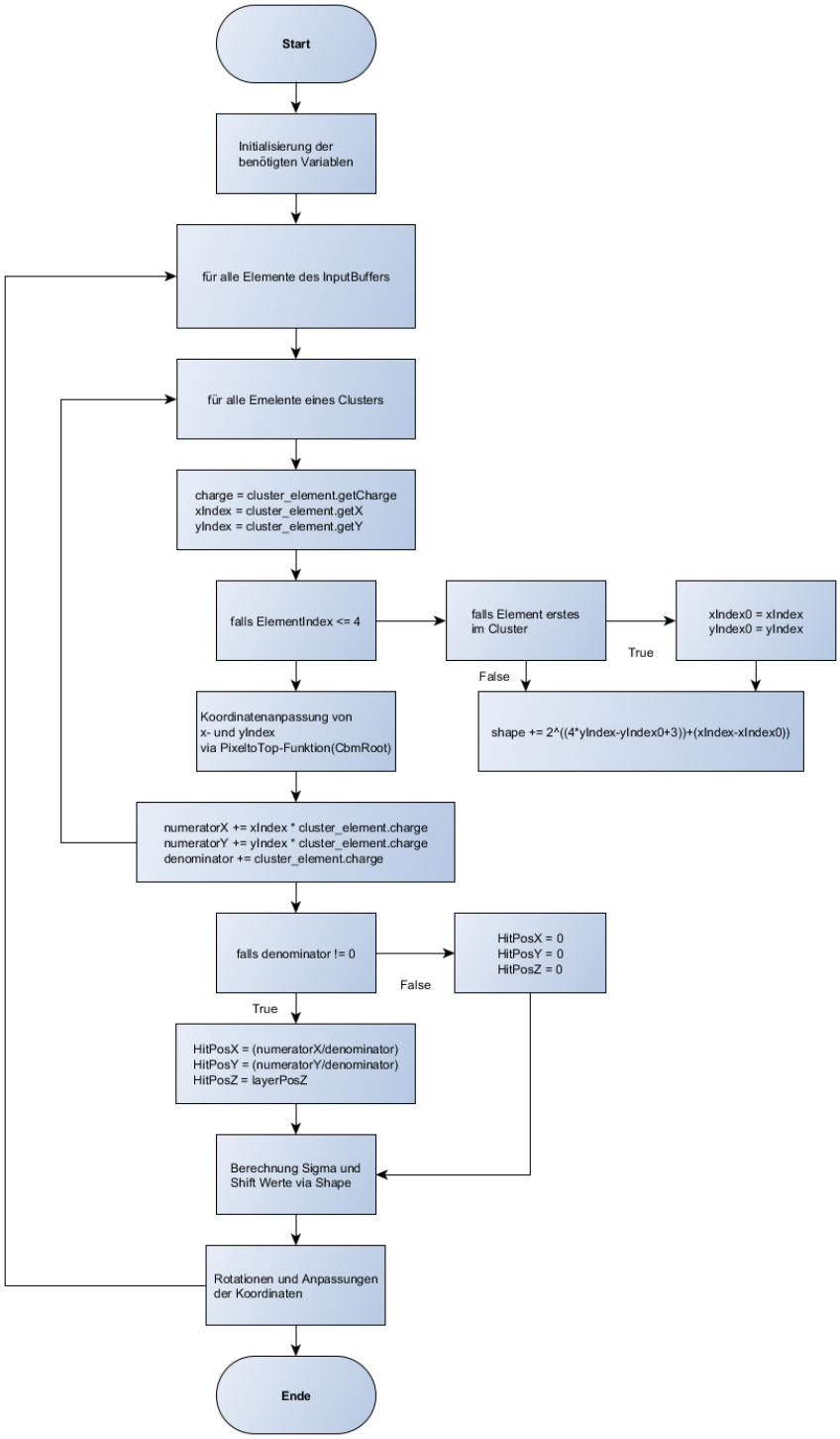

Abbildung 9: Ausgangsalgorithmus des Hitfinders

4 Optimierungen

4.1 Clusterfinder

Als ersten Schritt für die Optimierung des Clusterfinders wurden unnötige Variablen gelöscht und existierende optimiert. Anschließend wurden Datenstrukturen abgewandelt, um möglichst effizient zu arbeiten und unnötige Operationen zu vermeiden. Der letzte Schritt der Optimierung für den Clusterfinder ist die Überarbeitung des Algorithmus.

4.1.1 Variablenoptimierung

In diesem Abschnitt befassen wir uns mit dem Löschen und den Optimierungsmöglichkeiten bei der Initialisierung von Variablen.
Nach der Kontrolle der 50 initialisierten Variablen hat sich herausgestellt, dass nur 19 in diesem Task unter Verwendung standen. Somit konnten 31 der Variablen gelöscht werden, sodass viel allokierter Speicher freigegeben wird.
Des Weiteren wurden Compileroptimierungen, wie beispielsweise "const" und "constexpr" genutzt. "Const" beschreibt hierbei eine Variable mit einem anfänglichen Wert, der während der gesamten Laufzeit nicht wandelbar ist. Damit sind auf diesen Variablen nur Leseoperationen möglich.
"Constexpr" beschreibt eine Optimierung, die dem Compiler ermöglicht, Berechnungen, Variablen und Funktionen schon während der Compilierzeit komplett auszuwerten. Somit wird während der Laufzeit des Programms Zeit gespart, da diese Berechnungen schon vollständig getätigt wurden. Diese Optimierung konnte bei sechs der gegebenen Variablen durchgeführt werden.
Diese Optimierung wird exemplarisch an der Variable "AdcSteps" gezeigt:

$$const\ constexpr\ AdcSteps\{-1\};$$

4.1.2 Datenstrukturen

Bei der Optimierung der Datenstrukturen wurde versucht die gegebenen Datenstrukturen durch effizientere zu ersetzen. Hierbei wurde meist auf die gegebenen Datenstrukturen der C++ Standard Library (STL) zurückgegriffen, da diese für einige Fälle performanter arbeiteten, beziehungsweise besser für den Compiler zu optimieren sind.

Um die Speicherausnutzung zu minimieren und Cachemisses zu vermeiden, wurden Datenstrukturen, auf welche häufig zugegriffen wurde, durch kleinere

ersetzt. Nehmen wir exemplarisch das anfänglich verwendete ClusterArray, welches ein Standard-Vector der STL, bestehend aus vom Rootframework bezogenen Int_t war. Dieser wurde nun umgewandelt in einen Vector, bestehend aus einem Pair von "shorts". Ein "short" ist kleiner als der Int_t und wird somit leichter gecached als eine Datenstruktur aus vielen größeren Elementen. Des Weiteren wurde im Ausgangsalgorithmus die Größe des Int_t nicht ausgelastet, da es maximal zu einem Wert von $1152Pixel \cdot 576Pixel = 663552$ kommen würde. Der Int_t bietet 4 Byte an Speicher und würde somit maximal die Zahl $\frac{2^{4Byte \cdot 8Bit}}{2} - 1 = 2147483647$ im positiven Zahlenbereich speichern können.

Die als nächstes betrachtete Optimierung bezieht sich auf die DigiMap. Diese ist eine Map der STL, welche immer aus einem Wertepaar (Key-Value-Paar) besteht. Der Schlüssel ist in dem hier behandelten Programmabschnitt ein Paar aus den Koordinaten eines getroffenen Pixels. Der Value des Map-Elements ist der Index des Pixels in dem Inputbuffer.
Die Map ist ein sortierter Container. Da wir an dieser Stelle keinen sortierten Speicher benötigen, ist die Verwendung eines sortierten Containers unnötig. Des Weiteren kostet es jedes Mal beim Einfügen und Löschen eines Elements aus der Map Rechenzeit, um die Datenstruktur neu zu sortieren.
Aufgrund der Betrachtungsweise liegt es nahe eine zweidimensionale Matrix zu benutzen, welche zwar einen Overhead an Daten wegen der nicht genutzten Pixel inne hat, aber dennoch leichter zu verarbeiten ist. Diese wurde durch ein zweidimensionales Array der Größe 1154x578 mit "shorts" implementiert, welches anfänglich den Index an den Stellen mit den jeweiligen Koordinaten gespeichert hat. Es wird hier an beiden Achsen jeweils zwei Pixel mehr initialisiert als die eingentliche Detektorgröße, um später Randbedingungen nicht beachten zu müssen. Alle anderen nicht getroffenen Pixel wurden in dieser mit einer "-1" initialisiert. An dieser Stelle wird der Initialwert "-1" genutzt, da die Null ein valider Index der Liste sein kann. Des Weiteren werden hier keine Pointer genutzt, da diese in C++ eine fixe Größe von 32 Bits haben und somit ähnlich wie der Int_t mehr Speicher als nötig in Anspruch nehmen würden.

4.1.3 Algorithmus

In Erweiterung zur Datenstrukturoptimierung wurden ebenfalls einige Änderungen am Algorithmus vorgenommen wie in Abbildung 14 und Abbildung 15 schematisch dargestellt.
Wie im vorherigen Unterkapitel beschrieben, wird nun zu Anfang eine zweidimensionale Matrix initialisiert, die die Indices der jeweiligen Digis im In-

putbuffer gespeichert hat. Außerdem wird ein weiterer Standard Vector namens coordArray initialisiert, der parallel zum Inputbuffer aufgebaut ist und somit in gleicher Reihenfolge die Koordinatenpaare der getroffenen Pixel gespeichert hat.
Im Gegensatz zum Ausgangsalgorithmus wird nun nicht mehr über die Elemente Inputbuffer iteriert, sondern über die Elemente im CoordArray.

Zu Beginn wird kontrolliert, ob das gerade entnommene Element aus dem CoordArray im Grid ungleich "-1" ist. Tritt dieser Fall ein, so folgt die Speicherung der Koordinaten des Digis als Wertepaar im ClusterArray. Außerdem wird der entsprechende Wert an den betreffenden Koordinaten des Digis in der Matrix auf "-1" gesetzt, damit die nächsten Betrachtung diesen nicht nocheinmal berücksichtigt.
Nun wird wie im Ausgangsalgorithmus über die Elemente des ClusterArrays iteriert. Es wird für jedes Koordinatenpaar im ClusterArray geschaut, ob einer der benachbarten Pixel ebenfalls getroffen wurde. Tritt auch dieser Fall ein, so wird das Koordinatenpaar in das ClusterArray geschrieben und der Wert der betoffenen Stelle im Grid auf "-1" gesetzt, um eine wiederholte Benutzung auszuschließen.
Das beschreiben des Outputbuffers verläuft analog zum Ausgangsalgorithmus, indem zuerst alle gefundenen Digis in einer Map gespeichert werden und diese anschließend in den OutputBuffer übernommen wird.
Die Abbildungen 10, 11, 12 und 13 sie die minimale, maximale und durchschnittliche Ausführzeit des Clusterfinders dar. Zu Beginn des Projekts lag der Durchschnittswert der Minimum-Bias-Events bei ca. 0,007s. Durch die getätigten Optimierungen hat der Clusterfinder hierbei nun eine Ausführzeit pro Event von ca. 0,004s. Bei den zentralisierten Kollisionen lag der Durchschnitt ursprünglich bei ca. 0,025s. und nach der Optimierung bei ca. 0,016s. Daraus ergibt sich folgender Speedup:

$$Speedup(Mbias) \approx \frac{0,007}{0,004} = 1,75$$
$$Speedup(Centr) \approx \frac{0,025}{0,016} \approx 1.56$$

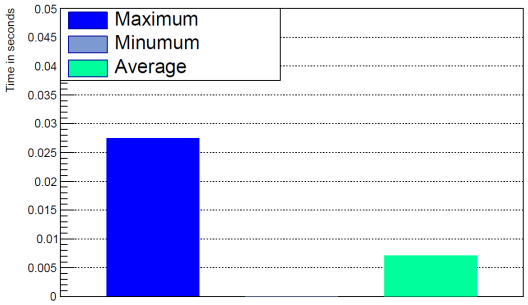

Abbildung 10: Clusterfinder Ausgangsalgorithmus mit Minimum-Bias-Kollision

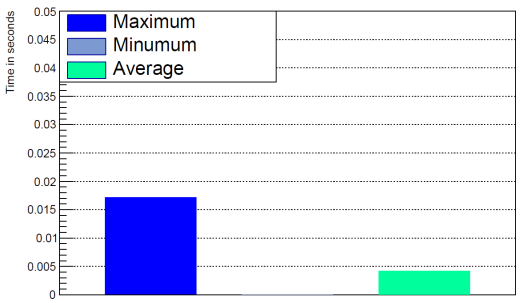

Abbildung 11: Clusterfinder angepasster Algorithmus mit Minimum-Bias-Kollision

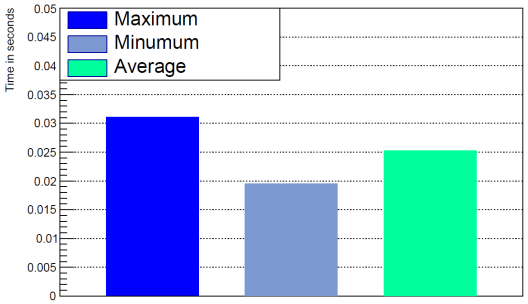

Abbildung 12: Clusterfinder Ausgangsalgorithmus mit Zentralisierten-Bias-Kollision

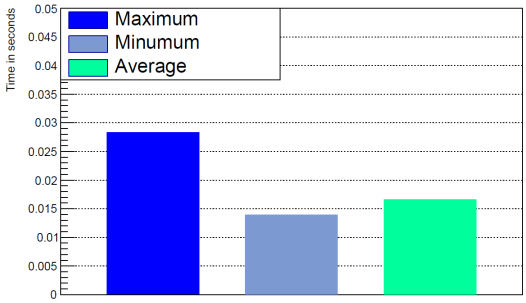

Abbildung 13: Clusterfinder angepasster Algorithmus mit Zentralisierten-Bias-Kollision

4.1.4 Parallelisierbarer Algorithmus

Da das Programm zukünftig auf einem Hochleistungsrechner laufen wird, kann es von Vorteil sein einen gut parallelisierbaren Algorithmus zu verwenden. Der derzeitige bietet diesen Spielraum nicht. Im folgenden Abschnitt wird erst kurz erläutert, warum der zuvor implementierte Algorithmus schlecht parallelisierbar ist. Anschließend folgt ein weiterer teils parallelisierter Algorithmus.

Die Parallelisierung des optimierten Codes gestaltet sich schwierig, da dieser nicht auf Multiprocessoring ausgelegt ist. Würden beispielsweise zwei Prozessoren gleichzeitig nach einem Cluster suchen, so würden diese niemals zu einem Zusammenschluss kommen, da es an dieser Stelle notwendig wäre, dass die unterschiedlichen Tasks voneinander wüssten. Somit entstünde eine Abhängigkeit. Diese Abhängigkeit führt dazu, dass der Algorithmus nicht immer

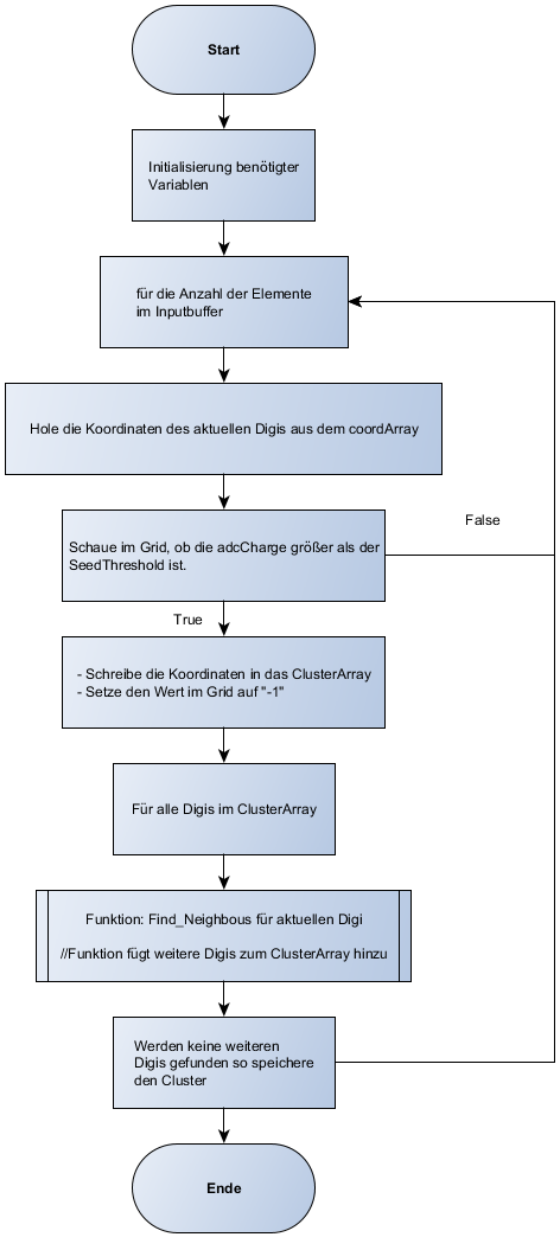

Abbildung 14: Neuer Clusterfinder

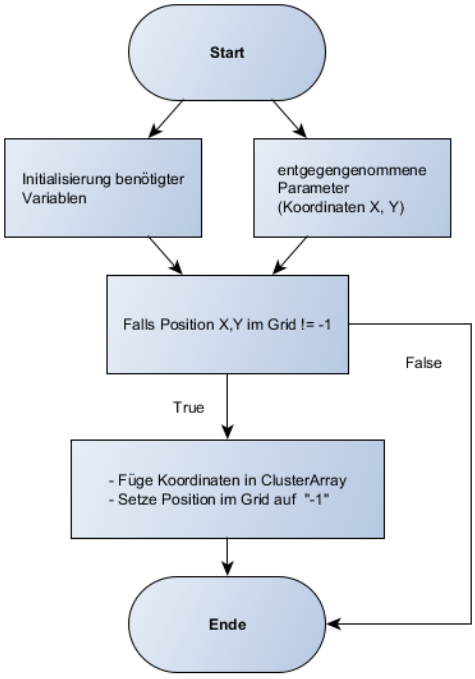

Abbildung 15: Neue FindNeighbours Funktion

eindeutig wäre und es von der Geschwindigkeit der einzelnen Tasks abhängig wäre, ob ein Digi nun zu einem Cluster gehört oder nicht.
Der neuen besser parallelisierbare Algorithmus beginnt analog zum vorherigen mit einem zweidimensionalen Array, welches an den Stellen der getroffenen Digis die Ladung (Charge) speichert und an ungenutzten Stellen eine 0 gespeichert hat. Des Weiteren wird eine neue Struktur mit dem Namen "clusterRow" definiert, welche aus einem Startzeiger, einem Endzeiger und drei Ladungswerten besteht. Außerdem wird ein weiteres zweidimensionales Array erzeugt, welches später Teilcluster speichert.
Eine schematische Datestellung des parallelisierbaren Abschnitts folgt in Abbildung 16 beschrieben. Er startet mit einer Schleife über alle Reihen der eingegebenen Matrix. Jede Zeile bekommt hierbei ihren eigenen Task und gewährleistet somit die Parallelität dieses Abschnitts. Innerhalb dieser Schleife wird eine weitere Schleife über die einzelnen Inhalte der Zeile gestartet. In dieser wird zu Anfang ein neues Element des Typs clusterRow initialisiert, welches zuvor in der gleichnamigen Struktur definiert ist. Anschließend wird eine Variable namens "next" definiert, welche den aktuell betrachteten Wert der Matrix speichert. Die Ausführung einer dritten Schleife läuft nun so lange ausgeführt, bis der nächste Wert rechts von dem betrachteten Wert ungleich Null ist. Für jeden gefundenen Wert ungleich Null werden die Ladungen und die gewichteten Ladungen aufaddiert. Stößt die Schleife auf eine "0", wird der letzte Wert als Endpointer des Clusters übernommen und dieser in das clusterArray an das Ende der jeweiligen Zeile geschrieben. Bis zu diesem Punkt kann der Algorithmus jede Zeile einzeln und unabhängig betrachten. Somit entsteht keine Datenabhängigkeit und er ist parallelisierbar.
Ab hier muss der Algorithmus linear weiterarbeiten, da es ansonsten zu Raceconditions kommen könnte, die die Ergebnisse beeinflussen und verfälschen würden. Dargestellt wird dieser Teil in Abbildung 17. Als nächstes werden drei Schleifen geschachtelt. Die Erste läuft über alle Elemente im clusterArray, die Zweite geht über die zuoberst liegende Reihe und die Dritte über die direkt Darunterliegende. Es wird nun verglichen, ob sich zwei Cluster, die zeilenweise direkt übereinander liegen, überschneiden. Tritt dieser Fall ein, so werden die Einzelladungen der Teilcluster aufaddiert. Ist dies nicht der Fall, so werden alle weiteren Teilcluster ebenfalls paarweise verglichen und gegebenenfalls aufaddiert, falls sich diese überschneiden sollten. Um Dopplungen in der Addition zu vermeiden, wird eine neue Struktur erzeugt, welche drei Werte enthält: Die Summe der Ladungen und jeweils die Summe der gewichteten Ladungen. Diese werden durch einen Pointer mit dem Teilcluster verknüpft. Die Ladungen werden somit immer nur dann aufaddiert, wenn der Teilcluster noch keinen Pointer auf eine Summe hat. Es wird entweder ein eigener neuer Cluster erzeugt oder er wird wie zuvor beschrieben bei Überschneidung Teil

eines bestehenden. Die Stärken dieses Algorithmus liegen hauptsächlich im ersten parallelisierbaren Teil. Der Zweite ist nicht auf mehrere Threads aufzuteilen, da, wenn zwei Cluster auf denselben Wert zugreifen und beide eine Addition zu tätigen haben, es vorkommen kann, dass ein Wert aufgrund von Raceconditions nicht geschrieben wird.
Der oben stehende Code zeigt seine Stärken bei vielen großen Clustern, weil dieser dort viel Zeit im Parallelteil gewinnt. Da die Daten aber meist aus vielen kleinen Clustern bestehen, lässt sich leicht abschätzen, dass der Code mehr Zeit benötigt, als der lineare Teil. Nach der Ausführung des parallelisierbaren Codes bestätigte sich diese Vermutung.
Dieses Ergebnis lässt sich ebenfalls leicht approximativ mathematisch begründen. Der lineare Code macht pro gefundenem Digi genau vier Schritte, um zu kontrollieren ob ein weiteres Digi im Umkreis von diesem liegt. n beschreibt hierbei die Anzahl der gefundenen Digis. i und j sind die Dimensionen des Grids und p die Anzahl der genutzten Prozessoren. Der bereits optimierte Algorithmus kontrolliert für jeden Digi genau vier Felder, um die Nachbarn zu finden. Somit ergibt sich approximativ eine Laufzeit von:

$$\mathcal{O}(4n)$$

Der parallelisierbare Algorithmus hingegen lässt sich approximativ im Worst Case in der folgenden Formel beschreiben:

$$\mathcal{O}(\frac{i}{2}^2 \cdot 2j + \frac{i \cdot j}{p})$$

Der linke Summand der Formel beschreibt den linearen Abschnitt des Algorithmus. Im Worst Case wurde die Detektorplatte an jedem zweiten Pixel getroffen. So würde sich ein Schachbrettmuster auf dem Grid ergeben und es wären nur Cluster mit der Größe "1" zu finden. Somit wären in einer Reihe genau $\frac{i}{2}$ Digis. Da der Algorithmus pro Reihe immer darüber und darunter kontrolliert, ob eine Überschneidung stattfindet, ergibt sich der Formelabschnitt $\frac{i^2}{2} \cdot 2j$. Der rechte Teil der Formel beschreibt den parallelen Abschnitt des Codes. Es werden i Pixel in jeder der j Zeilen des Grids auf p Prozessoren aufgeteilt. Da gilt, dass $n < i < j$. So lässt sich leicht sagen, dass der lineare Code echt schneller ist als der teils zu parallelisierende.

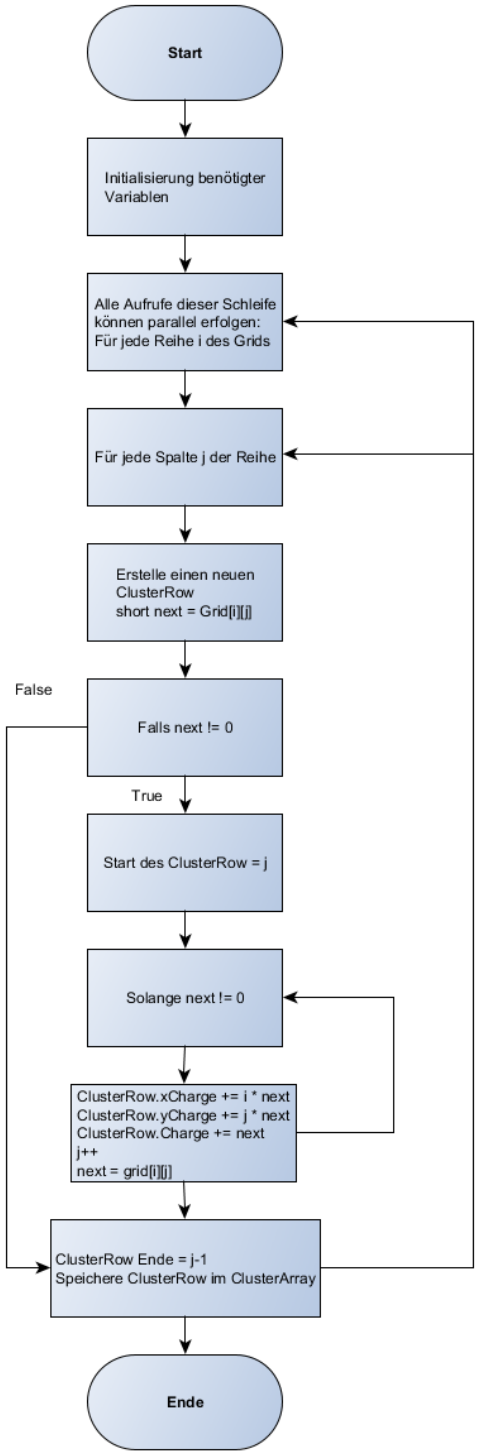

Abbildung 16: paralleler Teil des parallelisierten Algorithmus

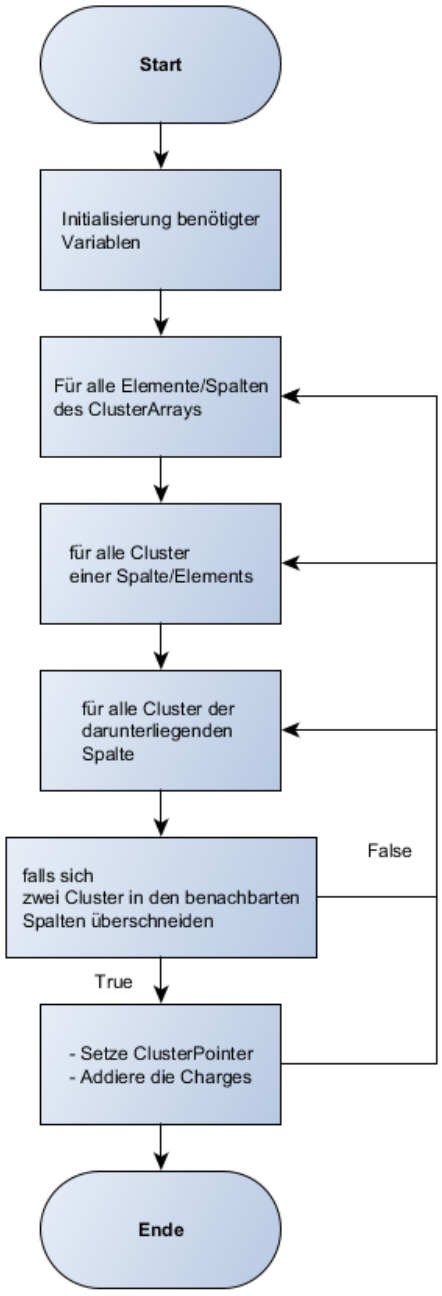

Abbildung 17: linearer Teil des parallelisierten Algorithmus

4.2 Hitfinder

Der folgende Abschnitt beschäftigt sich mit der Optimierung des Hitfinders. Dabei wird weniger auf die eigentliche Optimierung des Tasks eingegangen, vielmehr auf die Integration des Hitfinders in das Modul des Clusterfinders. Wie schon anfänglich beschrieben, wird der Hitfinder maßgeblich dazu verwendet, das gewichtete Mittel zu bilden und anschließend Rotationen zu berechnen, welche für die weitere Verarbeitung der Ergebnisse relevant werden. Da es nicht notwendig ist, diesen Programmabschnitt als einzelnen Task zu gestalten, wurde der gesamte Hitfinder-Algorithmus in den Clusterfinder integriert. Dieser Schritt wurde getätigt, da der Hitfinder maßgeblich mit dem Lesen von Daten des Clusterfinders und dem anschließenden Schreiben der Ergebnisse beschäftigt war. Des Weiteren bot sich eine Zusammenlegung der Tasks an, da bereits der Clusterfinder auf den Daten arbeitete, welche der Hitfinder für die Berechnungen benötigt. Durch die Integration werden daher die getätigten Lese- und Schreibzugriffe drastisch reduziert.
Die Integration des Hitfinders lässt sich im Allgemeinen in zwei Schritte aufteilen. Der erste Schritt ist das Extrahieren und Einfügen der Berechnung des gewichteten Mittelwertes in den Clusterfinder. Der zweite kleinere Schritt beschäftigt sich mit dem Aufbereiten der Daten für die Ausgabe des Hitfinders. Da in diesem Abschnitt ein Hybrid der beiden Tasks erzeugt wird, ist im weiteren Verlauf der Arbeit nicht mehr vom Cluster- und Hitfinder als einzelne Tasks die Rede, sondern von dem im Anschluss an dieses Kapitel entstandenen ”DigiToHit”-Task.

4.2.1 Berechnung gewichtetes Mittel

Das gewichtete Mittel ist eine stochastische Rechenvorschrift, um die Position des Hits im Cluster genau zu bestimmen.

$$X_{med} = \frac{\sum_{digi \in Cluster}(X_{digi} \cdot Q_{digi})}{\sum_{digi \in Cluster} Q_{digi}}$$
$$Y_{med} = \frac{\sum_{digi \in Cluster}(Y_{digi} \cdot Q_{digi})}{\sum_{digi \in Cluster} Q_{digi}}$$

X_{digi} bzw. X_{digi} bezeichnet hierbei die X-/Y-Koordinate des betrachteten Digis. Q_{digi} hingegen ist die Ladung, die an der Stelle des betrachteten Digis gemessen wurde. Die Formeln werden nun unterteilt. Zum einen die im Zähler des Bruchs befindlichen Summe und zum anderen die Summe, welche im Nenner definiert ist. Daher werden im Anschluss genau drei Summenberechnungen im neuen Task für diesen Abschnitt vorhanden sein:

$$1) \sum_{digi \in Cluster} (X_{digi} \cdot Q_{digi})$$
$$2) \sum_{digi \in Cluster} (Y_{digi} \cdot Q_{digi})$$
$$3) \sum_{digi \in Cluster} Q_{digi}$$

Wie bereits im Abschnitt 4.1.2 angesprochen liegt dort nun ein zweidimensionales Array vor, die zunächst noch Indices des Inputbuffers beinhaltet. Diese werden nun durch die Ladungen der dazugehörigen Digis ersetzt. Somit umgehen wir erstens den Umweg, immer auf den Inputbuffer zu referieren, zweitens sind schon alle Daten in dem Grid vorhanden, welche an dieser Stelle benötigt werden (Koordinaten und Ladung).
Die im Grid gespeicherte Ladung kann nun ohne weitere Umwege zum Inputbuffer oder anderen Datenstrukturen ausgelesen werden und jedes Mal, wenn ein neuer Digi einem Cluster hinzugefügt wird, direkt in die Summe der Ladungen aufgenommen werden. Der folgende Codeabschnitt zeigt die Lambda-Funktion, die aufgerufen wird, wenn ein neues Element des Clusters gefunden wird. Eine Lambda-Funktion ist eine Funktion, die in einer anderen Funktion definiert werden kann. Diese wird hier verwendet, um eine gute Struktur und Lesbarkeit des Codes zu gewährleisten. Hierbei wird die Ladung der gefundenen Digis direkt in die oben erläuterten Summenformeln mit eingerechnet.

Der folgende Codeabschnitt wird nicht weiter erläutert, da es sich hierbei nur um eine Umformung des bereits im Hitfinder verwendeten Codes handelt.
Bei der "«"-Operation handelt es sich um den Shift-Operator. Dieser verschiebt bitweise die eingegebene Zahl in Binärdarstellung nach links. Die Operation ermöglicht es hier Zweierpotenzen zu berechnen und diese anschließend aufzuaddieren. Lab fungiert hier nur als Ausgabedatenstruktur für die PixelToTop-Funktion.

```
auto calculate_median = [&](int channelX, int channelY,
Double_t charge){
    if( counter <= 4 ){
        if( counter == 0 ){
            xIndex0 = channelX;
            yIndex0 = channelY;
        }
        shape += 1 << ((4*(channelY-yIndex0+3))
        +(channelX-xIndex0));
    }
    fSensor->PixelToTop((channelX-1),(channelY-1),lab);

    x = lab[0];
    y = lab[1];
    numeratorX   += (x*charge);
    numeratorY   += (y*charge);
    denominator  += charge;
    counter++;
};
```

Listing 1: Code zur Mittelwertberechnung des DigiToHit-Tasks

4.2.2 Integration der Datenaufbereitung

Die Integration der Datenaufbereitung ist der kleinste Teil des gesamten Abschnitts, da diese in ihren Grundzügen keine Änderung erfährt. Sie bleibt bestehen, wie im Ausgangsalgorithmus (siehe 3) beschrieben, und wird im kombinierten Task aufgerufen, wenn ein Cluster komplett erstellt und somit keine weiteren Digis mehr zu diesem hinzugefügt werden. Die dann berechneten Hits werden anschließend in den OutputBuffer geschrieben, um von anderen Tasks weiterverarbeitet und analysiert zu werden.

4.3 Integration in den Programmkontext

Der neue Task ("DigiToHit" gesamte Code zum Task befindet sich im Anhang) muss nun noch in den Gesamtkontext der MVD Software integriert werden (siehe schematische Darstellung in Abbildung 18). Dazu wird in diesem Abschnitt erst erläutert, wie die Struktur der Software aufgebaut ist. Anschließend werden die notwendigen Änderungen beschrieben, die durchgeführt werden müssen, um den neuen Task erfolgreich in den Kontext einzubinden. Es wird sich mit dieser Thematik nur kurz auseinandergesetzt, da es sich hierbei nicht um die eigentliche Optimierung handelt.
Die Eingabedaten aus der Simulation werden zuerst den Detektorprogrammteil übergeben. Dieser ist sowohl bei dem MVD als auch bei allen anderen Detektoren dafür zuständig, die Eingabedaten auf die unterschiedlichen Tasks zu verteilen. Da der Programmcode des MVD maßgeblich die Daten einzelner Sensoren verarbeitet, wurde an dieser Stelle ein weiterer Programmabschnitt integriert, der die Eingabedaten auf die einzelnen SensorTasks verteilt. Die Klasse CbmMvdDetektor stellt Funktionen bereit, welche die Eingabedaten an die Sensortasks übergibt. Hierbei stehen Funktionen der Klasse CbmMvdSensor unter Verwendung, da der Programmabschnitt CbmMvdDetektor nur für die Weiterleitung an die Tasks zuständig ist und die Sensorunterteilung noch nicht betrachtet. Ausgeführt werden die Tasks in einem taskspezifischen Programmteil. Im Falle des neu erzeugten "DigiToHitTask" heißt dieser "CbmMvdDigiToHit". Für die Integration des neuen Tasks mussten demnach die Klassen "CbmMvdDetektor" und "CbmMvdSensor" so angepasst werden, dass sie die eingegebenen Digis nicht an den "CbmMvdClusterfinder" weiterleiten, sondern an den neu hinzugefügten "CbmMvdDigiToHit". Außerdem wurde die Weiterleitung der Daten an den Hitfinder aus der Struktur entfernt, da dieser wie in Abschnitt 4.2 erläutert, bereits im DigiToHitTask vorhanden ist. Das Übergeben der Daten an weitere Tasks nach der Verarbeitung des DigiToHit-Prozesses wurde ebenfalls angepasst, sodass diese nun die gleiche Struktur aufweist wie die Weiterleitung des vorherigen Hitfinder-

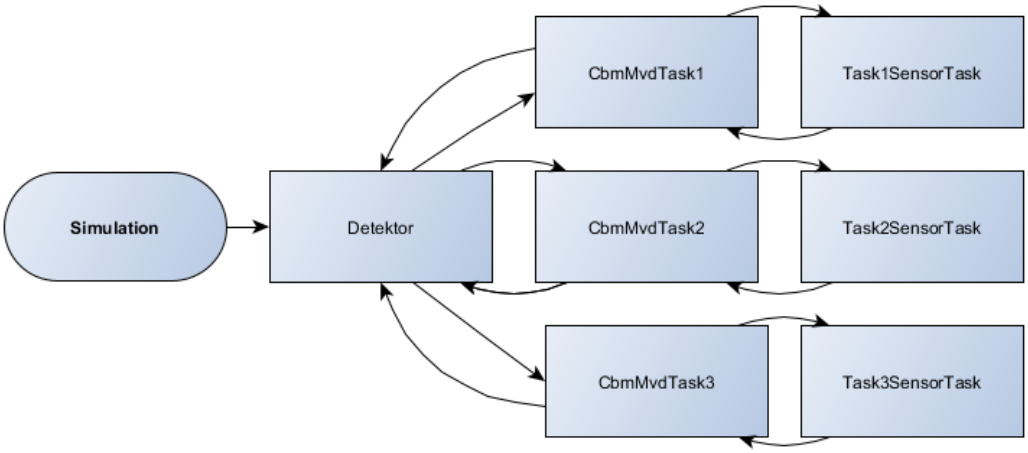

Abbildung 18: Schematische Darstellung des CbmMvd-Programm-Kontext

Tasks.
Zuletzt muss das Programm in die Cmake und Linkerstruktur eingebunden werden, sodass Compiler und Linker den neuen Task in den Programmkontext übernehmen.

Nach der erfolgreichen Integration des "DigiToHit"-Tasks in den Programm-Kontext lässt sich der folgende Speedup im Vergleich zum Clusterfinder erkennen (siehe Abbildung 20 und Abbildung 22):

$$Speedup(Mbias) \approx \frac{0,007}{0,003} = 2,3$$
$$Speedup(Centr) \approx \frac{0,025}{0,015} \approx 1,67$$

Da nun aber sowohl Cluster- als auch Hitfinder in einem Task zusammengeführt wurden, berechnet sich der schlussendliche Speedup durch das Hinzunehmen der Zeit des Hitfinders (siehe Abbildung 19 und Abbildung 12). Daraus ergibt sich ein ungefährer Gesamtspeedup:

$$Speedup(Mbias) \approx \frac{0,007 + 0,006}{0,003} \approx 4,34$$
$$Speedup(Centr) \approx \frac{0,025 + 0,012}{0,015} \approx 2,47$$

Man kann außerdem in Abbildung 23 und 24 deutlich einen linearen Anstieg der Bearbeitungszeit im Bezug auf die Anzahl der gefundenen Hits erkennen. Für Events bei denen wenige oder keine Hits gefunden wurden konvergiert die Bearbeitungszeit gegen Null.

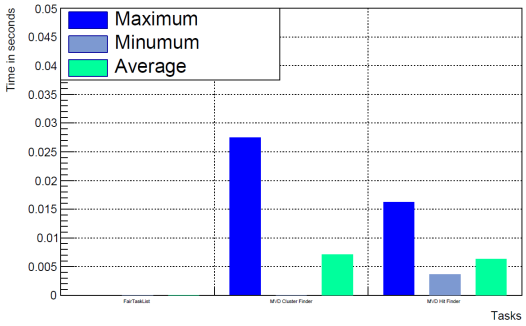

Abbildung 19: Cluster- & Hitfinder Ausgangsalgorithmen mit Minimum-Bias-Kollisionen

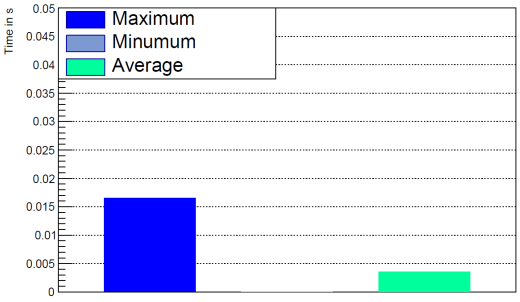

Abbildung 20: Clusterfinder angepasster Algorithmus mit Minimum-Bias-Kollisionen

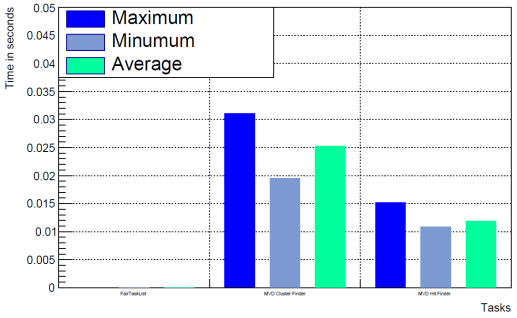

Abbildung 21: Cluster- & Hitfinder Ausgangsalgorithmen mit zentralisierten Kollisionen

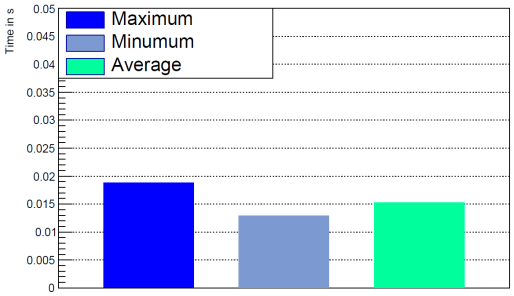

Abbildung 22: Clusterfinder angepasster Algorithmus mit zentralisierten Kollisionen

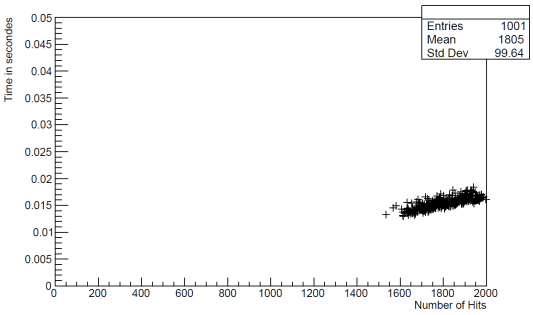

Abbildung 23: DigiToHit Zeit pro Event mit zentralisierten Kollisionen

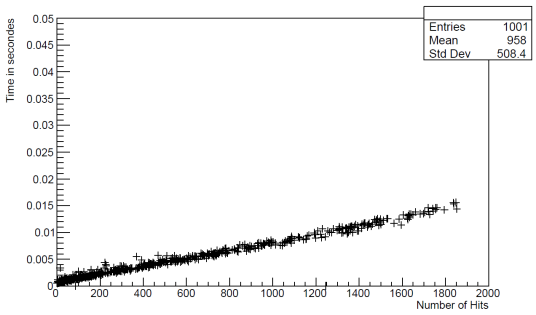

Abbildung 24: DigiToHit Zeit pro Event mit Minimum-Bias-Kollisionen

5 Ergebnis

Nach allen behandelten Optimierungen lässt sich feststellen, dass der Code signifikant zeitlich verbessert werden konnte. Im Vergleich zum anfänglichen Algorithmus wurde ein ungefährer Speedup von 4,34 bzw. 2,47 erreicht. Dieser lässt sich maßgeblich auf die Optimierung des Clusterfinder-Algorithmus, Datenstrukturen und das Zusammenlegen der Tasks zurückführen. Variablenanpassungen hatten hier nur einen geringen Einfluss auf die Performanz des Codes.
Die Anpassung des Algorithmus hat einen höheren Datenverbrauch zur Folge. Es bleibt noch die Frage offen, ob der Datenverbrauch des Grids (2-Dimensionale Matrix des Clusterfinders, welche die Ladungen der Digis beinhaltet) das Programm bei einer parallelen Ausführung für mehrere Sensoren gleichzeitig den Cache überfüllen würde und das Programm aufgrund von Cachemisses langsamer würde. Dieser Test konnte aufgrund von Schwierigkeiten mit der Mehrfachausführung des Codes leider nicht in dieser Arbeit dargelegt werden.
Das Zusammenführen von Cluster und Hitfinder erschwert zwar die Überprüfbarkeit der einzelnen Codeabschnitte, da hier die von vornherein gegebene Modularisierung unterbrochen wurde. Dafür kommt es jedoch zu einem großen Performanzanstieg. Es werden für die Performanz teure Schreib- und Leseoperationen umgangen und die notwendigen Berechnungen direkt beim ersten Zugriff auf die geforderten Daten getätigt.
Es lässt sich abschließend sagen, dass das so überarbeitete Programm eingesetzt werden kann wie es nun ist. Da sich die Spezifikationen und Bauweise des Detektors noch ändern können, wird dieser Programmabschnitt gegebenenfalls nur zu Simulations- und Analysezwecken verwendet, wird jedoch den Rekonstruktionsprozess dabei beschleunigen.

6 Zusammenfassung

Die hier vorliegende Bachelorarbeit befasst sich mit dem Thema der Optimierung der lokalen Rekonstruktion im Micro Vertex Detektor des CBM-Experiments. Dabei handelt es sich um eine Codeoptimierung in der Programmiersprache C++(11).
Der in dieser Arbeit betrachtete Code ist teil des Micro Vertex Detektor des CBM-Experiments, das am im Bau befindlichen FAIR in Darmstadt durchgeführt werden soll. Es werden im Laufe dieser Arbeit zwei Tasks vorgestellt und optimiert. Zu Anfang wird sich mit dem Clusterfinder auseinandergesetzt, der die Aufgabe hat zusammenhängende Strukturen in einem 2-dimensionalen Pixelfeld zu finden. Der zweite Task ("Hitfinder") berechnet anschließend Hits (das Gewichtete Mittel) innerhalb der gefundenen Strukturen.
Der Fokus dieser Arbeit liegt maßgeblich auf der zeitlichen Optimierung. Es wird sich dabei mit der Variablen-, Datenstruktur- und Algorithmenoptimierung auseinandergesetzt.
Die Arbeit beschreibt die Wahl des Themas und warum sich explizit mit der lokalen Rekonstrukton des MVD auseinandergesetzt wurde. Sie setzt Grundlagen zum weiteren Verständnis der Arbeit und erläutert anschließend die Schritte der Optimierung.

Literatur

[1] Johann-Wolfgang-Goethe-Universität Frankfurt am Main. Logo of goethe university frankfurt am main by adrian frutiger. https://commons.wikimedia.org/wiki/File:Logo-Goethe-University-Frankfurt-am-Main.svg, 2006. accessed on 2019-04-16.

[2] J De Cuveland, V Lindenstruth, CBM collaboration, et al. A first-level event selector for the cbm experiment at fair. In *Journal of physics: Conference series*, volume 331, page 022006. IOP Publishing, 2011.

[3] M Deveaux, S Amar-Youcef, C Dritsa, I Fröhlich, C Müntz, S Seddiki, J Stroth, T Tischler, and C Trageser. Design considerations for the micro vertex detector of the compressed baryonic matter experiment. *arXiv preprint arXiv:0906.1301*, 2009.

[4] Facility for Antiproton and Ion Research. Das compressed baryonic matter experiment bei fair. https://fair-center.eu/fileadmin/fair/experiments/CBM/documents/CBM_flyer_ToT.pdf. accesed on 2019-03-05.

[5] GSI Helmholtzzentrum für Schwerionenforschung GmbH. Fles group. https://www.gsi.de/work/forschung/cbmnqm/cbm/activities/fles.htm. accessed on 2019-03-05.

[6] GSI. Gsi cbmroot. https://redmine.cbm.gsi.de/projects/cbmroot, 2018. accesseed on 2019-04-13.

[7] GSI. Gsi fairroot. https://cbmroot.gsi.de/, 2018. accessed on 2019-04-13.

[8] Kitware. Cmake. https://cmake.org/overview/. accessed on 2019-03-05.

[9] Borislav Milanović. *Development of the Readout Controller for the CBM Micro-Vertex Detector*. PhD thesis, 2015.

[10] P. Senger and V. Friese. Cbm report 2012-01, nuclear matter physics at sis-100, the cbm collaboration. 2012.

[11] Tobias Tischler. *Mechanical Integration of the Micro Vertex Detector for the CBM Experiment*. PhD thesis, Master's thesis, Goethe-Universität Frankfurt, 2015. 19, 2015.

7 Anhang

7.1 Quellcode zum CbmMvdSensorDigiToHitTask

```
// -------------------------------------------------------------------------
// -----                  CbmMvdSensorDigiToHitTask source file        -----
// -----                  16.04.19 Edited by K. Hunold                 -----
// -------------------------------------------------------------------------



#include "CbmMvdSensorDigiToHitTask.h"
#include "TClonesArray.h"

#include "TObjArray.h"
#include "FairLogger.h"
#include <cstring>

#include "TGeoManager.h"
#include "TGeoTube.h"
#include "TArrayD.h"
#include "TObjArray.h"
#include "TRefArray.h"
#include <TMatrixD.h>


using std::cout;
using std::endl;
using std::pair;
using std::vector;
using std::map;

//Initializing Variables
Int_t dth_fAdcSteps(-1);
Int_t dth_fAddress(0);
Float_t dth_fAdcStepSize(-1.);
std::vector<TH1F*> dth_fPixelChargeHistos;
std::vector<TH1F*> dth_fTotalChargeInNpixelsArray;
const constexpr Int_t fCounter(0);
const constexpr Float_t fSigmaNoise(15.);
const float dth_fSeedThreshold = 1.;
const float dth_fNeighThreshold = 1.;
const constexpr Bool_t inputSet(kFALSE);
const constexpr Bool_t fAddNoise(kFALSE);
std::map<std::pair<Int_t, Int_t>, Int_t> dth_ftempPixelMap;

//2 bigger becuase we are also checking -1 and (range + 1) usually 576, 1152
const constexpr int pixelRows = 578;
const constexpr int pixelCols = 1154;

Float_t dth_grid[pixelCols][pixelRows];
std::vector<std::pair<short, short>> dth_clusterArray;
std::vector<std::pair<short, short>> dth_coordArray;
std::vector<int> dth_refIDArray;
const constexpr Int_t fGausArrayLimit = 5000;


TVector3     pos;
TVector3     dpos;
Double_t     lab[3];
Double_t     numeratorX;
Double_t     numeratorY;
Double_t     denominator;
Int_t        xIndex;
Int_t        yIndex;
Double_t     x,y;
Int_t        xIndex0;
Int_t        yIndex0;
int          ID              = 0;
int          counter         = 0;
UInt_t       shape           = 0;


// -----   Default constructor   -------------------------------------------
CbmMvdSensorDigiToHitTask::CbmMvdSensorDigiToHitTask()
  : CbmMvdSensorDigiToHitTask(0,0)
{
}
```

```
// -------------------------------------------------------------------------

// -------------------------------------------------------------------------
CbmMvdSensorDigiToHitTask::~CbmMvdSensorDigiToHitTask()
{
    if ( fOutputBuffer ) {
        fOutputBuffer->Delete();
        delete fOutputBuffer;
    }
}
// -------------------------------------------------------------------------

// -------------------------------------------------------------------------
 CbmMvdSensorDigiToHitTask::CbmMvdSensorDigiToHitTask(Int_t iMode, Int_t iVerbose)
  : CbmMvdSensorTask(),
        fAdcDynamic(200),
        fAdcOffset(0),
        fAdcBits(1),
    fDigiMap(),
    fDigiMapIt(),
    fVerbose(iVerbose),

    //HitFinder Variables
    fSigmaNoise(15.),
    fHitPosX(0.),
    fHitPosY(0.),
    fHitPosZ(0.),
    fHitPosErrX(0.0005),
    fHitPosErrY(0.0005),
    fHitPosErrZ(0.0)

{
}
// -------------------------------------------------------------------------

// -----   Virtual private method Init   -----------------------------------
void CbmMvdSensorDigiToHitTask::InitTask(CbmMvdSensor* mysensor) {


    fInputBuffer = new TClonesArray("CbmMvdDigi",100);
    fOutputBuffer = new TClonesArray("CbmMvdHit", 100);

    dth_fAdcSteps = (Int_t)TMath::Power(2,fAdcBits);
    dth_fAdcStepSize = fAdcDynamic/dth_fAdcSteps;

    fSensor = mysensor;
    dth_fAddress = 1000*fSensor->GetStationNr() + fSensor->GetSensorNr();

    std::memset(dth_grid, 0, sizeof(dth_grid));

    initialized = kTRUE;
}
// -------------------------------------------------------------------------

// -----   Virtual public method Reinit   ----------------------------------
Bool_t CbmMvdSensorDigiToHitTask::ReInit() {
    cout << "-I- " <<"CbmMvdSensorDigiToHitTask::ReInt----------------"<<endl;
    return kTRUE;
}
// -------------------------------------------------------------------------

// -----   Virtual public method ExecChain   ---------------
void CbmMvdSensorDigiToHitTask::ExecChain() {
  Exec();


}
// -------------------------------------------------------------------------

// -----   Public method Exec   ---------------
void CbmMvdSensorDigiToHitTask::Exec() {

    int nDigis = fInputBuffer->GetEntriesFast();
    if(nDigis > 0){

        fOutputBuffer->Delete();
        inputSet = kFALSE;
        short iDigi=0;

        CbmMvdDigi* digi = (CbmMvdDigi*) fInputBuffer->At(iDigi);

        if(!digi){
           cout << "-E- : CbmMvdSensorFindHitTask -"
```

```
            << "Fatal: No Digits found in this event."<< endl;
        }



        dth_clusterArray.reserve(nDigis);
        dth_coordArray.reserve(nDigis);
        dth_refIDArray.reserve(nDigis);


        //Initialize Grid and Arrays
        for(Int_t k=0;k<nDigis;k++){
            digi = (CbmMvdDigi*) fInputBuffer->At(k);

            if ( digi->GetRefId() < 0)
            {
              LOG(FATAL) << "RefID of this digi is -1" <<
                "this should not happend"<< FairLogger::endl;
            }

            //apply fNeighThreshold
            Float_t curr_digi_charge = digi->GetCharge();

            short dth_current_digi_X = digi->GetPixelX();
            short dth_current_digi_Y = digi->GetPixelY();
            dth_coordArray.emplace_back(std::make_pair(dth_current_digi_X, dth_current_digi_Y));


            if(curr_digi_charge >= dth_fNeighThreshold){
                //puts index into dth_grid.
                dth_grid[dth_current_digi_X + 1][dth_current_digi_Y + 1] = curr_digi_charge;
            }
        }

//_____________________________________________________________
        //Walk through all Digis in the coordArray
        for ( auto &curr_coord : dth_coordArray ) {

            auto &dth_current_digi_X = curr_coord.first;
            auto &dth_current_digi_Y = curr_coord.second;



            auto &root_digi_pos_charge =
             dth_grid[dth_current_digi_X + 1][dth_current_digi_Y + 1];

            //Apply Threshold
            if(GetAdcCharge(root_digi_pos_charge) >= dth_fSeedThreshold){

                pos = {0,0,0};
                dpos = {0,0,0};
                numeratorX          = 0;
                numeratorY          = 0;
                denominator         = 0;
                counter             = 0;

                xIndex0 = dth_current_digi_X - 1;
                yIndex0 = dth_current_digi_Y - 1;

//_____________________________________________________________

                dth_clusterArray.clear();
                dth_clusterArray.emplace_back(curr_coord);

                //Calculating Median for the first Element
                lab[0] = 0;
                lab[1] = 0;
                lab[2] = 0;

                shape &= 1 << ((4*(dth_current_digi_Y - 1
                 - yIndex0+3))+(dth_current_digi_X - 1 - xIndex0)) ;

                fSensor->PixelToTop((dth_current_digi_X - 1), (dth_current_digi_Y - 1), lab);

                numeratorX    += lab[0]*root_digi_pos_charge;
                numeratorY    += lab[1]*root_digi_pos_charge;
                denominator   += root_digi_pos_charge;

                root_digi_pos_charge = 0;
//_____________________________________________________________
                for (short i = 0; i < dth_clusterArray.size(); i++){

                    auto &index = dth_clusterArray[i];
```

```
                    //Lambda-Funktion to Find neighbours
                    auto checkNeighbour = [&](short channelX, short channelY){
                        auto &curr_digi_pos_charge = dth_grid[channelX + 1][channelY + 1];

                        if (curr_digi_pos_charge != 0){
                            lab[0] = 0;
                                    lab[1] = 0;
                                    lab[2] = 0;

                                    //Generating Shape
                                    if( counter <= 3 ){
                                        shape &= 1 << ((4*(channelY - 1 - yIndex0+3))
                                        +(channelX - 1 - xIndex0)) ;
                                    }

                                    fSensor->PixelToTop((channelX - 1), (channelY - 1), lab);
                                    //Calculating Sums for wheighted Median
                                    numeratorX    += lab[0]*curr_digi_pos_charge;
                                    numeratorY    += lab[1]*curr_digi_pos_charge;
                                    denominator   += curr_digi_pos_charge;
                                    counter++;

                                    //Saving current Digi in ClusterArray
                                    dth_clusterArray.emplace_back(
                                    std::make_pair(channelX, channelY));

                                    //Marking Digi in Grid as used/not relevant anymore
                            curr_digi_pos_charge = 0;
                        }
                    };

                    short channelX = index.first;
                    short channelY = index.second;

                    //Execute FindNeighbour-Funktion in every direction
                    checkNeighbour(channelX + 1, channelY);
                    checkNeighbour(channelX - 1, channelY);
                    checkNeighbour(channelX, channelY + 1);
                    checkNeighbour(channelX, channelY - 1);

                }

                //Compute Center of Gravity
                //_____________________________________________________________

                Double_t layerPosZ  = fSensor->GetZ();
                            Double_t sigmaIn[3], sigmaOut[3], shiftIn[3], shiftOut[3];

                            //Calculate x,y coordinates of the pixel in the laboratory ref frame
                            if(denominator!=0)
                            {
                                fHitPosX = (numeratorX/denominator);
                                fHitPosY = (numeratorY/denominator);
                                fHitPosZ = layerPosZ;
                            }
                            else
                            {
                                fHitPosX = 0;
                                fHitPosY = 0;
                                fHitPosZ = 0;
                            }


//_____________________________________________________________
                            switch(shape){
                                case 12288  : { sigmaIn[0]=0.00053; sigmaIn[1]=0.00063;
                                sigmaIn[2]=0.; shiftIn[0]=-0.00000; shiftIn[1]=-0.00001;
                                shiftIn[2]=0.; }
                                case 208896 : { sigmaIn[0]=0.00035; sigmaIn[1]=0.00036;
                                sigmaIn[2]=0.; shiftIn[0]=-0.00000; shiftIn[1]=-0.00002;
                                shiftIn[2]=0.; }
                                case 69632  : { sigmaIn[0]=0.00028; sigmaIn[1]=0.00028;
                                sigmaIn[2]=0.; shiftIn[0]=-0.00000; shiftIn[1]=-0.00002;
                                shiftIn[2]=0.; }
                                case 28672  : { sigmaIn[0]=0.00028; sigmaIn[1]=0.00039;
                                sigmaIn[2]=0.; shiftIn[0]=-0.00000; shiftIn[1]=-0.00001;
                                shiftIn[2]=0.; }
                                case 143360 : { sigmaIn[0]=0.00024; sigmaIn[1]=0.00022;
                                sigmaIn[2]=0.; shiftIn[0]=+0.00020; shiftIn[1]=+0.00008;
                                shiftIn[2]=0.; }
                                case 200704 : { sigmaIn[0]=0.00024; sigmaIn[1]=0.00022;
                                sigmaIn[2]=0.; shiftIn[0]=-0.00020; shiftIn[1]=-0.00011;
```

```
                                shiftIn[2]=0.; }
                                case 77824   : { sigmaIn[0]=0.00024; sigmaIn[1]=0.00022;
                                sigmaIn[2]=0.; shiftIn[0]=-0.00020; shiftIn[1]=+0.00008;
                                shiftIn[2]=0.; }
                                case 12800   : { sigmaIn[0]=0.00024; sigmaIn[1]=0.00022;
                                sigmaIn[2]=0.; shiftIn[0]=+0.00020; shiftIn[1]=-0.00011;
                                shiftIn[2]=0.; }
                                case 4096    : { sigmaIn[0]=0.00027; sigmaIn[1]=0.00092;
                                sigmaIn[2]=0.; shiftIn[0]=+0.00002; shiftIn[1]=+0.00004;
                                shiftIn[2]=0.; }
                                default      : { sigmaIn[0]=0.00036; sigmaIn[1]=0.00044;
                                sigmaIn[2]=0.; shiftIn[0]=-0.00000; shiftIn[1]=-0.00002;
                                shiftIn[2]=0.; }
                            }
//_____________________________________________________________

                            // Consider Sensor Orientation

                            TGeoHMatrix*     RecoMatrix = fSensor->GetRecoMatrix();
                            TGeoHMatrix      RotMatrix;
                            RotMatrix.SetRotation(RecoMatrix->GetRotationMatrix());

                            RotMatrix.LocalToMaster(sigmaIn, sigmaOut);
                            RotMatrix.LocalToMaster(shiftIn, shiftOut);

                            fHitPosX+= shiftOut[0];
                            fHitPosY+= shiftOut[1];
                            fHitPosZ+= shiftOut[2];

                            pos.SetXYZ(fHitPosX,fHitPosY,fHitPosZ);
                            dpos.SetXYZ(TMath::Abs(sigmaOut[0]),
                             TMath::Abs(sigmaOut[1]), TMath::Abs(sigmaOut[2]));

                            //Create Hit
//_____________________________________________________________

                Int_t indexX, indexY;

                            Double_t local[2];
                            local[0] = pos.X();
                            local[1] = pos.Y();

                            fSensor->TopToPixel(local, indexX, indexY);
                            Int_t nHits = fOutputBuffer->GetEntriesFast();

                            new((*fOutputBuffer)[nHits])
                            CbmMvdHit(fSensor->GetStationNr(), pos,
                             dpos, indexX, indexY, ID, 0);
                            CbmMvdHit* currentHit = (CbmMvdHit*) fOutputBuffer->At(nHits);
                            currentHit->SetTime(fSensor->GetCurrentEventTime());
                            currentHit->SetTimeError(fSensor->GetIntegrationtime()/2);
                            currentHit->SetRefId(ID);
                            ID++;

            }

        }

        Int_t nHits = fOutputBuffer->GetEntriesFast();

        dth_clusterArray.clear();
        fInputBuffer->Delete();
        dth_coordArray.clear();
    }
}


// ---------------------------------------------------------------------------
short CbmMvdSensorDigiToHitTask::GetAdcCharge(Float_t curr_charge)
{
    int adcCharge;

    if(curr_charge<fAdcOffset){return 0;};

    adcCharge = int( (curr_charge-fAdcOffset)/dth_fAdcStepSize );

    if ( adcCharge>dth_fAdcSteps-1 ){
        return dth_fAdcSteps-1;
    } else {
        return adcCharge;
    }
}
```

```
//-----------------------------------------------------------------------
void CbmMvdSensorDigiToHitTask::Finish() {
//taken from Clusterfinder 03.12.2014 by P. Sitzmann
    if(fShowDebugHistos){
        cout << "\n================================================================" << endl;
        cout << "-I- " << GetName() << "::Finish: Total events skipped: " << fCounter << endl;
        cout << "================================================================" << endl;
        cout << "-I- Parameters used" << endl;
        cout << "Gaussian noise [electrons]     : " << fSigmaNoise << endl;
        cout << "Noise simulated [Bool]         : " << fAddNoise << endl;
        cout << "Threshold seed [ADC]           : " << dth_fSeedThreshold << endl;
        cout << "Threshold neighbours [ADC]     : " << dth_fNeighThreshold << endl;
        cout << "ADC - Bits                     : " << fAdcBits << endl;
        cout << "ADC - Dynamic [electrons]      : " << fAdcDynamic << endl;
        cout << "ADC - Offset [electrons]       : " << fAdcOffset << endl;
        cout << "================================================================" << endl;


        TH1F* histo;
        TH2F* clusterShapeHistogram;
        TCanvas* canvas2=new TCanvas("HitFinderCharge","HitFinderCharge");
        canvas2->Divide(2,2);
        canvas2->cd(1);
                if(fChargeArraySize<=7){
                clusterShapeHistogram= new TH2F("MvdClusterShape", "MvdClusterShape",
                 fChargeArraySize, 0, fChargeArraySize,fChargeArraySize, 0, fChargeArraySize);
                for (Int_t i=0;i<fChargeArraySize*fChargeArraySize; i++) {
                                histo = dth_fPixelChargeHistos[i];
                                Float_t curr_charge= histo->GetMean();
                                clusterShapeHistogram->Fill(
                                i%fChargeArraySize,i/fChargeArraySize,curr_charge);
                        }
                }
                clusterShapeHistogram->Draw("Lego2");
                canvas2->cd(2);
                histo= dth_fPixelChargeHistos[50];
                histo->Draw();
                canvas2->cd(3);
                histo= dth_fPixelChargeHistos[51];
                histo->Draw();
            canvas2->cd(4);
    }
}
//-----------------------------------------------------------------------
ClassImp(CbmMvdSensorDigiToHitTask)
```

Listing 2: CbmMvdSensorDigiToHitTask.cxx Gesamt Code

Danksagung

An dieser Stelle möchte ich mich bei allen Bedanken, die mich im Laufe dieser Arbeit unterstützt haben.
Zuerst möchte ich mich bei Herrn Prof. Dr. Lindenstruth bedanken, der diese Arbeit betreut und begutachtet hat.

Des weiteren bedanke ich mich bei Dr. Andreas Redelbach, der für die direkte Unterstützung und Betreuung dieser Arbeit zuständig war.
Außerdem möchte ich mich gerne bei Annegret Hunold und Johanna Weiser bedanken, die für die sprachliche Korrekthiet dieser Arbeit gesorgt haben.