GOETHE UNIVERSITÄT
FRANKFURT AM MAIN

Fachbereich 12 Informatik und Mathematik
Institut für Informatik

Bachelorarbeit

Optimierung der lokalen Rekonstruktion des STS-Detektors am CBM-Experiment

Florian Boeck
Studiengang: Bachelor Informatik
Matrikelnummer: 6085388

Frankfurt am Main
6. Juni 2019

Eingereicht bei:
Prof. Dr. Volker Lindenstruth
Frankfurt Institute for Advanced Studies

Danksagungen

An dieser Stelle möchte ich mich beim Lehrstuhl Hochleistungsrechnerarchitektur der Goethe-Universität bedanken, mir die Möglichkeit gegeben zu haben, mit meiner Bachelorarbeit aktiv an aktuell relevanter Forschung teilnehmen zu dürfen.
Zudem möchte ich mich bei Dr. Sergey Gorbunov bedanken, der mir geholfen hat, meine Idee zur Reduzierung der Kombinatorik im Hit Finder zu erweitern und mich bei der Implementierung dieser unterstützt hat.
Weiterer Dank gilt den Mitarbeitern der GSI, Dr. Volker Friese und Dr. Florian Uhlig, die sich die Zeit genommen haben, mir bei einigen Fragen zur Implementierung beizustehen. Auch für die Möglichkeit der Teilnahme an den CBM-Meetings möchte ich mich bedanken.
Abschließend gilt mein größter Dank Dr. Andreas Redelbach, der mir ausführlich die Problemstellung meiner Bachelorarbeit und die Relevanz dieser erklärt hat. Er hat mich während der gesamten Bachelorarbeit begleitet, mir beim Aufsetzen des Projekts geholfen und ständig neue Ideen und Vorschläge zur Verbesserung gegeben. Aufkommende Fragen zu physikalischen Grundlagen im CBM-Experiment wurden mir verständlich erklärt. Darüber hinaus bekam ich die Möglichkeit, den Prototypen des CBM-Experiments vor Ort zu besichtigen.

Abstract

Das Compressed Baryonic Matter Experiment (CBM-Experiment) ist ein von der Facility for Antiproton and Ion Research (FAIR) entwickeltes Teilchenbeschleunigerexperiment, welches das Quantenchromodynamik-Phasendiagramm (QCD-Phasendiagram) mit Hilfe von hochenergetischen Zusammenstößen von Atomkernen erforscht. Bei diesen Kollisionen entstehen viele neue Teilchen, deren Untersuchung Rückschlüsse auf die Entstehung und auf den weiteren Verlauf des Universums ziehen lassen [1]. Um die Teilchen zu detektieren und zu analysieren werden eine Vielzahl von Detektorsystemen im CBM-Experiment eingesetzt.
Das Silicon Tracking System (STS) ist ein Detektorsystem von zentraler Bedeutung für das CBM-Experiment. Seine Aufgabe besteht darin, den Verlauf der entstandenen geladenen Teilchen zu rekonstruieren und den Impuls dieser zu messen [2]. Dazu sind hinter dem Target mehrere Stationen des STS in verschiedenen Abständen aufgestellt. Die aus dem Kollisionsereignis entstandenen geladenen Teilchen treffen auf die einzelnen Stationen des STS und liefern auswertbare Messungen [2].
Diese Messungen werden in einem zweischrittigen Prozess, der lokalen Rekonstruktion, verarbeitet. Im ersten Schritt werden die einzelnen Messungen aufgrund ihrer zeitlichen und örtlichen Nähe zu sogenannten Clustern gruppiert. Im zweiten Schritt werden die Cluster zu einzelnen Treffern mit den Stationen, sogenannten Hits, kombiniert [2].
Im laufenden Betrieb erfasst das CBM-Experiment Daten in einer Rate von bis zu 1 TB/s, wovon ein wesentlicher Teil auf das STS zurückzuführen ist [2, 3]. Diese Datenmenge ist in der Praxis nicht abspeicherbar, weshalb die Daten kontinuierlich in Echtzeit gefiltert und verarbeitet werden müssen [3].
Diese Arbeit befasst sich mit der Herausforderung der Echtzeit-Datenverarbeitung. Durch Verbesserung des Programmcodes in seiner sequentiellen Form und durch eine ermöglichte Parallelisierung konnte eine Optimierung der lokalen Rekonstruktion des STS erreicht werden. Dazu wurden die einzelnen Schritte der lokalen Rekonstruktion auf Effizienz, Schnelligkeit und Datenabhängigkeiten untersucht und anschließend für eine erhöhte Performanz optimiert. Des Weiteren wurden Datenstrukturen und Algorithmen angepasst, um eine Parallelisierung zu ermöglichen und größere Datenmengen schneller verarbeiten zu können.
Insgesamt wurde durch die Optimierung ein signifikanter Performanzgewinn in der sequentiellen Ausführung erzielt. Obwohl durch die Parallelisierung ein zusätzlicher Aufwand entstand, konnte eine Skalierung mit der Kern- bzw. Threadanzahl erreicht werden. Das bringt die lokale Rekonstruktion des STS einen Schritt weiter in Richtung kontinuierliche Echtzeit-Datenverarbeitung.

Inhaltsverzeichnis

Abkürzungsverzeichnis

CBM	Compressed Baryonic Matter
Cluster	Gruppierung von Digis anhand ihrer zeitlichen und örtlichen Nähe
DAQ-System	Data Acquisition System, erhält die Messungen der Detektoren und bildet aus diesen Digis
Digi	Messung mit digitalisierter Spannung, Kanaladresse und Zeitstempel aus einem Siliziumsensor
FAIR	Facility for Antiproton and Ion Research
FLES	First-level Event Selector
GSI	Gesellschaft für Schwerionenforschung
Hit	Aus Clustern rekonstruierter Wechselwirkungspunkt mit einer Station des STS
Kollisions-ereignis	Ursprüngliche Kollision zwischen Schwerionenstrahl und Target des CBM-Experiments
QCD	Quantenchromodynamik
SIS100	Schwerionensynchrotron 100
STS	Silicon Tracking System
Track	Spur eines geladenen Teilchens durch die verschiedenen Stationen des STS
Wechsel-wirkungsrate	Anzahl der Kollisionsereignisse pro Zeiteinheit

Abbildungsverzeichnis

1 Einleitung

Die Teilchenphysik spielt in der heutigen Zeit eine entscheidende Rolle, um unser Verständnis über Natur und Universum weiter zu fundieren. Oftmals werden physikalische Theorien über die Existenz bestimmter Teilchen aufgestellt, ohne einen experimentellen Nachweis zu besitzen. Teilchenbeschleunigeranlagen versuchen dort anzusetzen. Durch Beobachtungen der Kollisionen von verschiedenen Atomkernen sollen sowohl Theorien überprüft, als auch Mechanismen und Auswirkungen dabei entstehender Teilchen erforscht werden.
Für die Auswertung der physikalischen Experimente, bei denen immense Mengen an Daten anfallen, reichen leistungsstarke Computer alleine nicht mehr aus [4]. Mindestens ebenso wichtig ist das effiziente Filtern der Messdaten und die schnelle Verarbeitung dieser, da oftmals das Speichern aller Messdaten einen großen, in der Praxis nicht realisierbaren Speicher erfordert [4, 5]. Für eine effiziente Auswertung der Messdaten ist es wichtig, die Leistung und den Speicher der vorhandenen Hardware maximal auszunutzen und eine optimale Auswertungskette der Daten zu implementieren.
Vor dieser Herausforderung steht das CBM-Experiment der Teilchenbeschleunigeranlage der GSI in Darmstadt. Dort wird innerhalb der nächsten Jahre das Compressed Baryonic Matter Experiment aufgebaut, um das QCD Phasendiagramm weiter zu erforschen [1]. Damit aussagekräftige Ergebnisse mit dem CBM-Experiment erzielt werden können, müssen die produzierten Messdaten effizient verarbeitet und analysiert werden, um so aus der produzierten Datenmenge relevante Kollisionsereignisse herauszufiltern zu können.
Im CBM-Experiment stellt dies eine besondere Herausforderung dar, denn die Daten werden bei einer Rate von bis zu 1 TB/s produziert, wodurch sie unmöglich abspeicherbar sind [3]. Bei anderen Experimenten wie dem ALICE-Experiment (A Large Ion Collider Experiment) liegt die produzierte Datenmengen lediglich bei 100 MB/s, bei größeren Experimenten wie dem ATLAS-Experiment (A Toroidal LHC Apparatus) werden 320 MB/s generiert [6].
Durch die hohe Datenproduktionsrate ist eine kontinuierliche Echtzeit-Verarbeitung der Daten im CBM-Experiment unausweichlich und erfordert performante Algorithmen und optimale Verarbeitungsketten [3].
Das STS ist ein Detektor von zentraler Bedeutung im CBM-Experiment, das durch seine direkte Platzierung hinter dem Kollisionsziel einen Großteil der Daten produziert [2]. Eine Optimierung der Verarbeitungskette und des Programmcodes des STS

trägt einen wesentlichen Anteil zur Verbesserung des gesamten CBM-Experiments bei.

1.1 Problemstellung

Ziel dieser Arbeit ist es, die Performanz des STS durch optimieren der Verarbeitungskette und des Programmcodes zu verbessern. Die Anforderungen an das Silicon Tracking System sind eine hohe Genauigkeit der Messdaten bzw. der daraus resultierenden Ergebnisse und eine damit einhergehende hohe Wechselwirkungsrate von bis zu 10 MHz [2, 3]. Aus der hohen Wechselwirkungsrate folgt eine hohe Datenproduktionsrate des STS [2]. Diese Datenmenge muss möglichst effizient gefiltert und verarbeitet werden, um eine Echtzeit-Verarbeitung garantieren zu können [3].
Dafür gibt es hauptsächlich zwei verschiedene Möglichkeiten.
Die erste Möglichkeit wäre die Anschaffung von mehr Hardware bzw. besserer Hardware. Weil die Daten des CBM-Experiments in einem dedizierten Rechenzentrum, dem Green IT Cube, verarbeitet werden, ist die Anschaffung von mehr Hardware räumlich nur begrenzt möglich [7]. Abgesehen davon sind die Anschaffungs- und Betriebskosten von mehr Hardware der ausschlaggebende Faktor. Sowohl die Anschaffung moderner, leistungsstarker CPUs und GPUs, als auch die zur Vernetzung benötigte Hardware tragen erheblich zu den Gesamtkosten des Projektes bei. Dabei ist das Budget zu berücksichtigen und Anschaffungen abzuwägen. Zudem ist auch die Hardware in ihrer Leistung beschränkt und die leistungsstärkste Hardware nützt nichts bei einem ineffizienten Programm.
Die zweite Möglichkeit stellt einer Verbesserung der Software dar. Durch die Vermeidung oder Optimierung von ineffizienten Programmabschnitten kann eine deutlich verbesserte Performanz erzielt werden. Zudem lassen sich oft Datenstrukturen verwenden, die eine einfachere und schnellere Benutzung der Daten erlauben. Dazu kommt, dass heutige Prozessoren mehrere Kerne und Threads besitzen, wovon die meisten nicht voll ausgenutzt werden oder sogar unbenutzt bleiben. Eine Parallelisierung der Software kann so zu einer deutlichen Performanzsteigerung führen. Mehr dazu in Abschnitt 2.6 [8]. Zusätzlich könnte eine Vektorisierung, also das volle Ausnutzen der einzelnen Prozessorkerne mit ihrem kompletten Speicher, angestrebt werden. So würden mehrere Elemente mit nur einer Anweisung, auch SIMD (Single Instruction Multiple Data) genannt, verarbeitet werden, was ebenso zur einer Performanzsteigerung führen würde [9].
In Anbetracht dieser Aspekte wurde in dieser Arbeit der Fokus auf eine generelle Optimierung des bestehenden Programmcodes sowie eine experimentelle Parallelisierung gelegt. Es wurde der gesamte Programmablauf betrachtet und auf Verbesserungspotentiale untersucht. Langsame Datenstrukturen wurden durch effizientere ersetzt und Umstrukturierungen wurden eingeführt, um Datenabhängigkeiten zu

eliminieren. Dadurch wurde eine Grundlage für die Parallelisierung gelegt und diese experimentell erforscht. Möglicherweise lässt sich der Ansatz Parallelisierung auf Grafikkarten übertragen. Dies ist jedoch nicht Bestandteil dieser Arbeit.

2 Grundlagen

2.1 Compressed Baryonic Matter Expriment

Das CBM-Experiment von FAIR, welches am GSI Helmholtzzentrum für Schwerionenforschung in Darmstadt entsteht, dient der Erforschung des QCD-Phasendiagramms in der Region hoher baryonischer Dichte [1, 10]. Es werden hochenergetische Kern-Kern-Kollisionen und Proton-Kern-Kollisionen bei Kerndichten von Neutronensternen untersucht und analysiert [1]. Dazu ensteht der neue Teilchenbeschleuniger SIS100, der an den bisherigen Teilchenbeschleuniger SIS18 anschließt und diesen erweitert [3]. Das SIS100 Synchrotron wird vorraussichtlich 2024 fertiggestellt, was zu der gleichzeitigen Inbetriebnahme des CBM-Experiments führt [3]. Der Strahl des SIS100 für das CBM-Experiment wird aus Metallionen, beispielsweise Goldionen, bestehen und in einer Energiespanne von 2 AGeV bis 11 AGeV operieren[3]. Das SIS100 Synchrotron und der Entstehungsort des CBM-Experiments sind in Abbildung 2.1 zu erkennen. Die Messungen werden mit einer hohen Wechselwirkungsrate von bis zu 10 MHz durchgeführt, um möglichst genaue Ergebnisse zu erhalten [1]. Die hohe Wechselwirkungsrate erfordert schnelle, robuste Detektorsysteme und eine neue Auslese- und Analysemethode, die mit dem ständigen Datenstrom von ungefähr 1 TB/s umgehen kann [3].
Insgesamt besteht das CBM-Experiment aus verschiedenen Detektorsystemen (siehe Abb. 2.2), die alle verschiedene Aspekte der Kollisionen analysieren. Das CBM-Experiment wird in zwei verschiedenen Konfigurationen benutzt, die für die Untersuchung von verschiedenen Teilchen geeignet sind. In Abbildung 2.2 ist die Elektron-Hadron Konfiguration zu sehen. Die Myonen Konfiguration tauscht den RICH (Ring Imaging Cherenkov) Detektor mit dem MUCH (Muon Chamber) Detektor aus und der ECAL (Electromagnetic Calorimeter) Detektor wird ersatzlos entfernt [3]. Das Silicon Tracking System wird in beiden Konfigurationen eingesetzt.

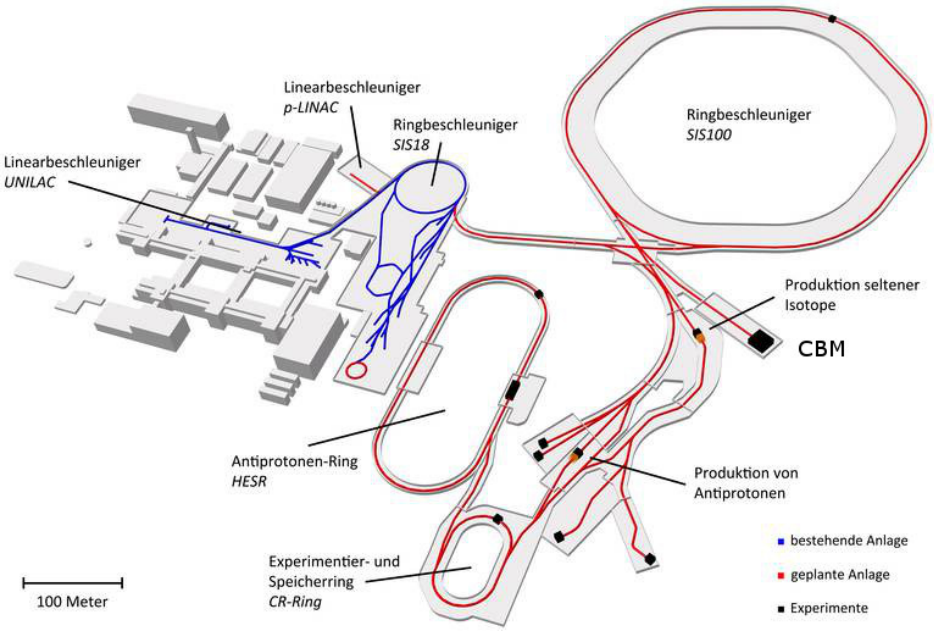


Abbildung 2.1: Beschleunigeranlage von FAIR und GSI [11]

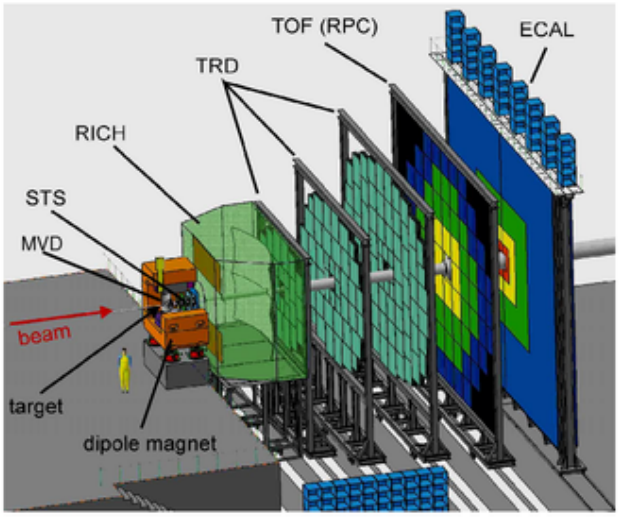


Abbildung 2.2: Detektorsysteme des CBM-Experiments in der Elektron-Hadron-Konfiguration[3]

2.2 Silicon Tracking System

Das STS ist ein zentraler Detektor im CBM-Experiment und wird zur Rekonstruktion der Spuren der geladenen Teilchen aus dem Kollisionsereignis zwischen Strahl und Target verwendet [2]. Eine weitere Aufgabe des STS ist die Impulsmessung dieser Teilchen, wobei die Genauigkeit in der Größenordnung von $\Delta p/p = 1\%$ liegt [2]. Um diese Aufgaben verwirklichen zu können liegt die räumliche Auflösung des Detektors bei $\lesssim 20\ \mu m$[3].
Das STS wird eigens von der GSI für das CBM-Experiment entwickelt, um allen Anforderungen und Aufgaben zu entsprechen[2]. Es besteht aus insgesamt 8 verschiedenen Stationen, die in einer Entfernung zwischen 30 cm und 100 cm zum Target innerhalb eines Dipolmagneten (siehe Abbildung 2.2 und 2.5 links) hintereinander angereiht sind [3]. Der Dipolmagnet wird zur Impulsmessung benötigt, da das erzeugte Magnetfeld die Spuren der geladenen Teilchen krümmt, wodurch schließlich der Impuls dieser berechnet werden kann [3]. Je langsamer die Teilchen sind, desto gekrümmter ist die Spur [3].
Die 8 Stationen des STS bestehen aus ca. 900 einzelnen Modulen, welche wiederum aus insgesamt ca. 1200 Silziumsensoren bestehen, die dem Detektor seinen Namen geben [2]. Die Siliziumsensoren werden in 3 verschiedenen Größen verwendet, je nach Ort und Lage des Moduls [2]. Jedes Modul besitzt ca. 2000 Kanäle mit denen die Ladungen der Teilchen gemessen werden [2]. Insgesamt gibt es dadurch ca. 1.8 Millionen Kanäle [2]. Die einzelnen Siliziumsensoren messen doppelseitig und die einzelnen Kanäle auf den Sensoren sind in einem Abstand von 58 μm angeordnet, was eine sehr genaue räumliche Auflösung garantiert [2]. Der Aufbau einer solchen Station aus Modulen und Siliziumsensoren ist in Abbildung 2.4 zu sehen. Die Anzahl aller verwendeten Komponenten wird in Abbildung 2.3 aufgelistet, wobei die Stationen ab dem Kollisionsziel durchnummeriert sind. Wie in der Abbildung zu erkennen ist, steigt mit zunehmender Entfernung vom Kollisionsziel die Größe der Station und somit auch die Anzahl der verwendeten Sensoren an.
Die Besonderheit an den Siliziumsensoren ist, dass sie bei einer Ladungsmessung selbstauslösend eine Nachricht an das Data Aquisition (DAQ) System schicken. Diese Nachricht enthält die gemessene Ladung, einen Zeitstempel und die Kanaladresse, in der die Messung stattfand [12].

Station	Leitern	Module	Sensoren	Auslesechips	Kanäle
1	8	76	76	1216	156k
2	12	100	100	1600	205k
3	12	108	132	1728	222k
4	14	116	144	1856	238k
5	14	112	168	1792	230k
6	14	112	168	1792	230k
7	16	136	216	2176	279k
8	16	136	216	2176	279k
Gesamt	106	896	1220	14336	1835k

Abbildung 2.3: STS Stationen mit ihren Komponenten [2]

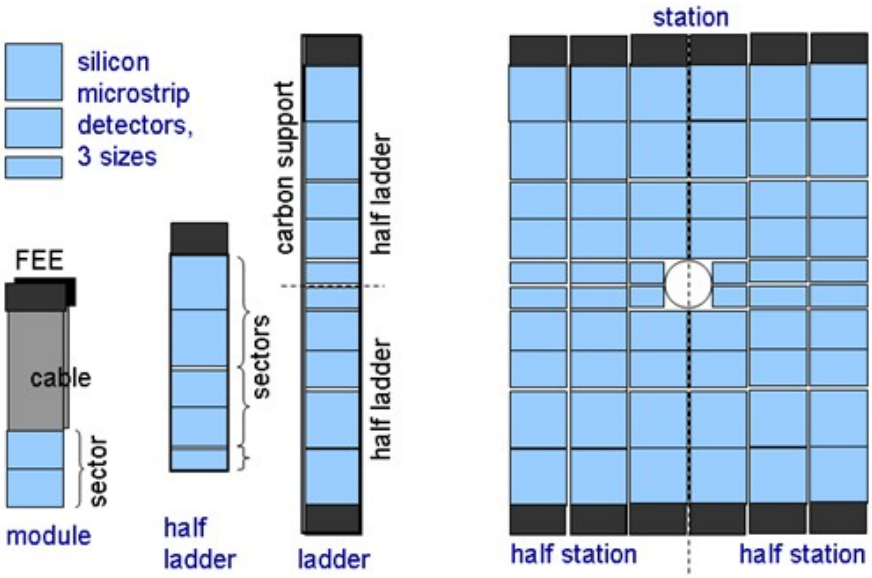


Abbildung 2.4: Aufbau einer Station des STS mit all seinen Komponenten [2]

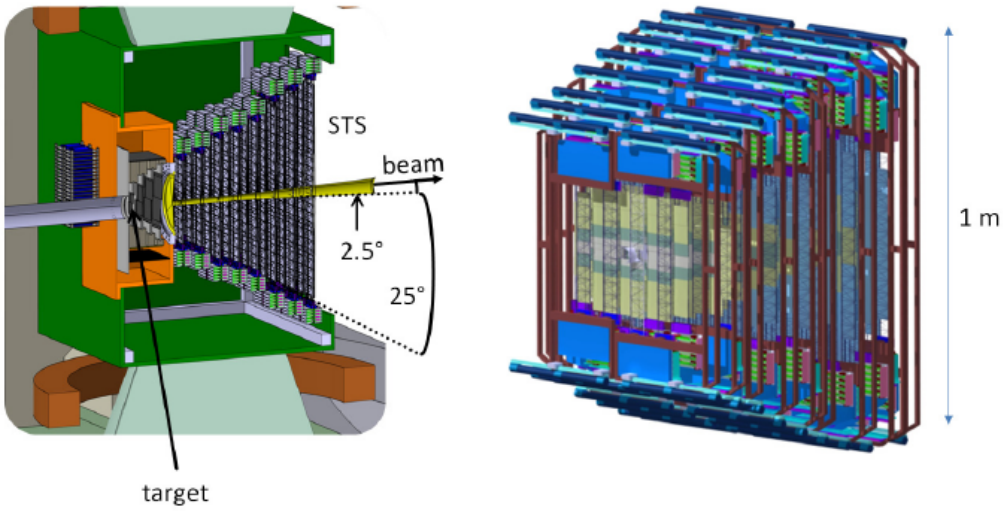


Abbildung 2.5: Links: Querschnitt der vorderen Detektorsysteme, das STS im Dipolmagneten (grüner Kasten) angebracht sind [13]
Rechts: Das STS mit seinen 8 Stationen [13]

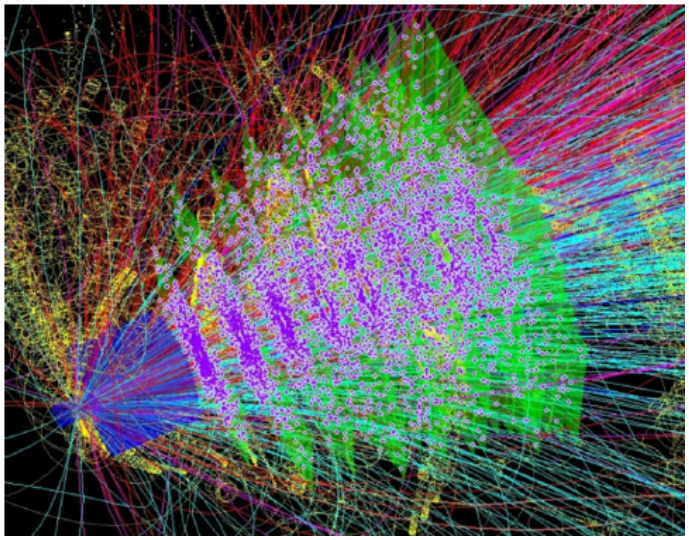


Abbildung 2.6: Graphische Darstellung der rekonstruierten Tracks durch die Hits (violette Punkte) mit den STS Stationen (grüne Scheiben) [13]

2.3 First-level Event Selector

Der First-level Event Selector (FLES) ist das zentrale Selektiersystem des CBM-Experiments, das aus den gemessenen Messdaten die wichtigen Kollisionsereignisse aussortiert, um diese später zu speichern [14]. Diese Prozedur erfordert schnelle Algorithmen zur Berechnung der Kollisionen und zur Rekonstruktion der Teilchenbahnen [14]. Insgesamt muss der FLES die produzierte Datenmenge von 1 TB/s so verarbeiten, dass sinnvolle und aussagekräftige Kollisionen bei einer Rate von 1 GB/s abgespeichert werden können [14]. Dafür ist der FLES auf eine Skalierbarkeit mit den vorhandenen Rechenknoten angesetzt, damit eine effiziente Verarbeitung gewährleistet ist. [14].
Um die Anforderungen für die verschiedenen Varianten des CBM-Experiments zu erfüllen, benutzt der FLES einen zweischrittigen Prozess, das Rekonstruieren der Kollisionsereignisse und das Selektieren dieser.
Im ersten Schritt, der Rekonstruktion der Kollisionsereignisse, werden zuerst die Messdaten aufbereitet. Die verschiedenen Detektoren messen hauptsächlich Ladungen, die zusammen mit Zeitstempel und Kanalinformation der Messung als Nachricht, auch Digi genannt, an das DAQ-System geschickt werden [12]. Die vom DAQ-System erhaltenen Nachrichten werden anhand ihres Zeitstempels in Zeitintervalle, sogennante Time Slices, zusammengefasst, die zum Berechnen der Kollisionsereignisse verwendet werden. Die Digis in einer Time Slice werden zudem nach ihrem Zeitstempel sortiert. Um keine Informationen zu verlieren, werden die Digis am Anfang und am Ende einer Time Slice zudem in der vorhergehenden und anschließenden Time Slice benutzt. So überlappen sich die Time Slices und können parallel auf verschiedenen Rechenknoten verarbeitet werden, da keine weiteren Datenabhängigkeiten mehr bestehen [14].
Ein passendes Zeitintervall bzw. eine passende Größe der Time Slices muss experimentell bestimmt werden, wobei sie jedoch größtmöglich sein sollte, um nicht zu viel Zeit mit Lese- und Schreiboperationen zu verbringen. Aktuell wird eine Größe von 1000 zugrundeliegenden Kollisionsereignissen pro Time Slice angestrebt [14].
Im zweiten Schritt werden von den erzeugten Kollisionsereignissen die Aussagekräftigsten selektiert und abgespeichert.
Es wurde eine Verarbeitung in Time Slices gewählt, da dies einige Vorteile gegenüber der kontinuierlichen Abarbeitung der Digis bietet [14]. So können direkt im zweiten Schritt Time Slices selektiert und abgespeichert werden, statt erst die relevanten Digis eines Kollisionsereignisses bündeln zu müssen. Auch für die Verarbeitung sind Time Slices mit einer uniformen Größe vorteilhaft, weil jeder Rechenknoten einen circa gleich großen Input erhält. Dadurch sind die Verarbeitungszeiten der Time Slices ähnlich und eine gute Lastaufteilung ist möglich.
Neben der Verwendung von GPUs werden zusätzlich Programmierschnittstellen, wie

OpenCL, zur parallelen Berechnung auf den heterogenen Systemen benutzt [14]. Da die Effizienz des FLES unerlässlich für die Resultate des CBM-Experiments ist, werden Teile des schon bestehenden Codes umgeschrieben, um eine erhöhte Performanz zu erreichen [14].

2.4 CbmRoot

Die Software für das CBM-Experiment wird im CbmRoot Framework implementiert, das auf dem FairRoot Framework basiert. Das FairRoot Framework wurde eigens für FAIR Experimente an der GSI Darmstadt entwickelt [15, 16]. Die Implementierungen von FairRoot für verschiedene Experimente sind in Abbildung 2.7 gezeigt. Das FairRoot Framework ist ein objektorientiertes Simulations-, Rekonstruktions- und Datenanalyse-Framework, das wiederum auf dem vom CERN entwickelten ROOT Framework basiert [15]. Ziel des ROOT Frameworks ist es, einen modularen Werkzeugkasten für Physikexperimente zur Verfügung zu stellen, mit dem sich diese als Software implementieren lassen [17]. Es bietet daher alle nötigen Voraussetzungen, um mit großen Datenmengen und deren statistischer Analyse, Visualisierung und Abspeicherung umzugehen [17]. Das FairRoot Framework erweitert das ROOT Framework um zusätzliche Kerneigenschaften, die für die Simulation von Detektoren sowie die Offline- und Online-Analyse der Daten benötigt werden [15]. Zudem enthält es noch einige nützliche Features wie z.B. die Spurvisualisierung [15]. Ein Beispiel dieser Spurvisualisierung für das STS ist in Abbildung 2.6 gezeigt.
Eine hilfreiche Funktion der Frameworks ist die Möglichkeit, die Analyse der Daten in einzelne Schritte, sogenannte Tasks, aufzuteilen. Für jedes Detektorsystem des CBM-Experiments gibt es eigene Tasks, die die Analyse und Funktionen des Detektorsystems implementieren. Durch die Aufteilung in Tasks entstehen modulare Einheiten, die nach Belieben zur Gesamtanalyse hinzugefügt oder entfernt werden können und somit ein individuelles Testen der Systeme ermöglichen. In FairRoot werden diese Tasks letztlich von der FairRun Instanz ausgeführt, welche der Ausgangspunkt jeder Ausführung ist [18].
Die aktuelle Version von CbmRoot bietet viele verschiedene Möglichkeiten zur Analyse der berechneten Daten oder Überprüfung der Ergebnisse. Zum Beispiel können die Daten mit Monte-Carlo-Simulationen weiter analysiert oder die Verteilungen der berechneten Ergebnisse angesehen werden. Während der Entwicklungsphase kann dadurch überprüft werden, ob die berechneten Ergebnisse den Erwartungen entsprechen.

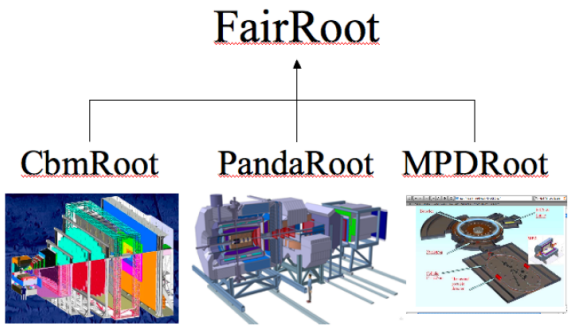


Abbildung 2.7: FairRoot Implementierungen für verschiedene Experimente [19]

2.5 Lokale Rekonstruktion

Die lokale Rekonstruktion im STS bezeichnet das Auffinden von Kollisionspunkten, auch Treffer oder Hits genannt, geladenener Teilchen mit den einzelnen Stationen des STS [2]. Diese Kollisionen werden schließlich zu Spuren der geladenen Teilchen kombiniert, die für die Impulsmessung der Teilchen benötigt werden [2]. Das Kombinieren der Kollisionen zu Spuren gehört nicht mehr zur lokalen Rekonstruktion. In Abbildung 2.6 sind die 8 Stationen des STS als grüne Platten veranschaulicht, die Hits auf diesen als violette Punkte und die daraus resultierenden Spuren als gekrümmte rote und türkise Linien.
Die für die lokale Rekonstruktion verwendeten Daten sind die Time Slices, die vom FLES erstellt werden. Diese enthalten die einzelnen Digis mit den gemessenen Ladungen der STS Stationen. Die Time Slices werden in einem zweischrittigen Prozess, der lokalen Rekonstruktion, verarbeitet.
Im ersten Schritt werden die Digis einer Time Slice aufgrund ihrer Zeit und ihrer Kanalinformation, in welchem Sensor sie gemessen wurden, zu Clustern gruppiert. Dafür wird vom Clustering Algorithmus die Vorsortierung der Digis nach ihrem Zeitstempel und die Zuordnung der Digis zu ihren Modulen ausgenutzt. Von den erzeugten Clustern werden anschließend noch einige Parameter berechnet, wie zum Beispiel ihr Ladungsschwerpunkt oder die durchschnittliche Zeit der Digis. [2].
Im zweiten Schritt, der Hit Rekonstruktion, werden aus den erzeugten Clustern dann Hits gebildet. Dabei besteht ein Hit aus mindestens 2 Clustern, einem von der Vorderseite und einem von der Rückseite einer Station [2]. Die Zeiten und Positionen jedes Clusterpaars werden verglichen, um zu überprüfen, ob dieses einen Hit ergibt.
Die lokale Rekonstruktion im STS ist Teil des ersten Schrittes des FLES und muss daher hochperformant sein. Die Optimierung der lokalen Rekonstruktion des STS ist ein Resultat aus dieser Anforderung des FLES.
Weiteres zu den einzelnen Schritten wird in Kapitel 3 erklärt.

2.6 OpenMP

OpenMP [20] ist eine Sammlung von Anweisungen, Variablen und Bibliotheken für Compiler, die es erlaubt Programme auf höchster Ebene zu parallelisieren. OpenMP dient dazu, Programmabschnitte auf verschiedene Prozessorkerne zu verteilen, sodass diese unabhängig voneinander laufen können. Die Parallelisierung kann in verschiedenen Formen erzeugt werden. Die einfachste Form ist das parallele Ausführen eines ganzen Code-Segments. Weitere Formen sind die Schleifen-Parallelisierung, die verschachtelte Parallelisierung und die Task-Parallelisierung, in welcher kleine Aufgaben deklariert und parallel ausgeführt werden.
Die Parallelisierung von Programmen erzeugt oftmals eine Vielzahl von Problemen, da sich alle Kerne bzw. Threads einer CPU die gleichen Ressourcen teilen. Auf Speicheradressen, die zwischen den Prozessorkernen geteilt sind, kann in einer zufälligen Reihenfolge von den individuellen Prozessorkernen zugegriffen werden. Dadurch kann es vorkommen, dass verschiedene Prozessorkerne auf der gleichen Speicheradresse operieren und sich überschreiben. Dieser Fall heißt Race Condition, da nicht absehbar ist, welcher Prozess wann auf welche Ressource zugreift und es so zu unvorhersehbaren Ergebnissen kommt, die allein von der zeitlichen Ausführung der Prozesse abhängen [21]. Diese neuen Probleme lassen sich durch bedachte Planung des Programms und/oder der Verwendung anderer Hilfsmittel beheben. Meist werden sogenannte Locks verwendet, die einen bestimmten Abschnitt oder eine bestimmte Speicheradresse nur für einen Prozessorkern zulassen. Andere Optionen sind das Definieren von kritischen Abschnitten oder einer Barriere [20]. In einem kritischen Abschnitt kann immer nur ein Prozessorkern gleichzeitig arbeiten. An einer Barriere muss jeder Prozessorkern warten, bis alle Prozessorkerne diese Barriere erreicht haben.
Damit eine gute Performanz mit der Parallelisierung durch OpenMP erreicht wird, sollten möglichst viele Operationen und Prozessorkerne unabhängig voneinander laufen können. Ebenso sollten generell wenige gemeinsame Speicher benutzt werden, auf denen Daten geschrieben und gespeichert werden.
Um herauszufinden, welcher theoretische Speedup durch eine Parallelisierung möglich ist, kann das Ahmdalsche Gesetz verwendet werden [22]. Dieses ist wie folgt definiert:

$$Speedup(n) = \frac{1}{r_s + \frac{r_p}{n}} = \frac{T_1}{T_p} \tag{2.1}$$

wobei r_s der prozentuale Anteil des Programm ist, der sequentiell ausgeführt werden muss und r_p der prozentuale Anteil des Programms ist, der parallelisiert werden

kann. Für beide Anteile gilt $r_s + r_p = 1$ und n bezeichnet die Anzahl an Prozessorkernen. Je größer der sequentielle Anteil des Programms ist, desto weniger effizient ist der Einsatz von mehr Prozessorkernen.
Weiterhin gilt, dass der Speedup durch den Quotienten aus Laufzeit auf einem Prozessor, T_1, und Laufzeit auf n Prozessoren, T_n, definiert ist.

3 Ausgangspunkt der lokalen Rekonstruktion

Ausgangspunkt der Arbeit war das CbmRoot Projekt der GSI [23]. Dort ist das gesamte Codeprojekt, das momentan für das CBM-Experiment entwickelt wird, implementiert. Da sich CbmRoot von FairRoot ableitet, ist der generelle Progammablauf über die Klasse `FairRun`, die Teil des FairRoot Frameworks ist, abgewickelt. Der Programmablauf über `FairRun` für die Rekonstruktionskette des STS wird in Abbildung 3.1 dargestellt.
Generell gibt es zwei verschiedene Varianten der lokalen Rekonstruktion, die während der Entwicklungsphase verwendet werden, wobei die erste ausschließlich zum Zwecke der Entwicklung vorhanden ist und im späteren Betrieb des CBM-Experiments nicht nutzbar sein wird.
In der ersten Variante, werden die simulierten Kollisionsevent (kurz Events) rekonstruiert, wobei die Zuordnung von Digis zu Events aus der Simulation verwendet wird. Diese Variante heißt eventbasierte lokale Rekonstruktion, da die Events aus den ihnen zugehörigen Digis rekonstruiert werden. Es wird ein Zusammenhang zwischen Digis und Events ausgenutzt, der mehr Überprüfungsmöglichkeiten bezüglich der Genauigkeit der Algorithmen ermöglicht, den es aber im späteren Experiment nicht gibt.
In der zweiten Variante werden die einzelnen Digis der Detektoren vom FLES in Time Slices gruppiert. Dies entspricht der tatsächlichen Messung und Verarbeitung der Daten während des Experiments im späteren Betrieb. Innerhalb dieser Time Slices werden dann aus den Digis Cluster und schließlich Hits gebildet. Diese Art der Rekonstruktion wird auch als zeitbasierte Rekonstruktion bezeichnet.
Ein entscheidender Faktor wird später das Bilden dieser Time Slices sein, denn von ihrer Größe hängt im Wesentlichen die Genauigkeit und Effizienz des Experiments ab. Wenn eine zu kleine Größe der Time Slices gewählt wird, werden möglicherweise zusammenhängende Digis auf verschiedene Time Slices verteilt. Oder es sind nicht genug Digis in einer Time Slice für eine aussagekräftige Rekonstruktion vorhanden. Ebenso ist die Größe entscheidend für die Software, denn mit größeren Time Slices steigt die Kombinatorik in manchen Programmteilen, wie dem Finden der Hits, quadratisch an. Das Finden einer passenden Größe der Time Slices ist Aufgabe des FLES und nicht Teil dieser Arbeit.
In dieser Arbeit wurde der Fokus auf die Optimierung der lokalen Rekonstruktion

mit der zweiten Variante gesetzt, da diese der späteren Implementierung und Verwendung des CBM-Experiments entspricht.
Für die Entwicklung standen außerdem 2 Datensätze aus simulierten Messungen zur Verfügung. Der Erste enthält zentrale Kollisionsereignisse, welche von Messrauschen bereinigt sind und somit eine Idealisierung darstellen. Der zweite Datensatz enthält Kollisionsereignisse mit Messrauschen und repräsentiert die echten Daten, wie sie im späteren Experiment erzeugt werden. Dieser Datensatz wird Minimum-Bias Datensatz genannt.
Für die Optimierung und Tests wurde der Minimum-Bias Datensatz gewählt und liegt allen folgenden Grafiken zugrunde.

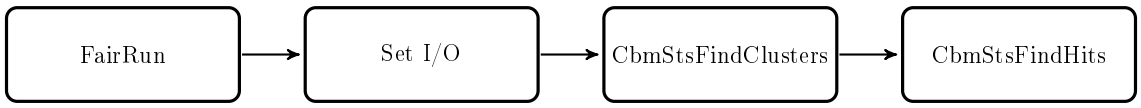


Abbildung 3.1: Progammablauf der lokalen Rekonstruktion

Hauptentwickler der Software für die lokale Rekonstruktion des STS ist Dr. Volker Friese von der GSI. Von ihm stammen die meisten relevanten Programmklassen zur Rekonstruktion der Cluster und Hits. Seine C++-Klassen wurden von mir unter Beachtung der Laufzeit genauer untersucht und anschließend in Zusammenarbeit mit Herrn Friese verbessert und angepasst.
Die in CbmRoot implementierten C++-Klassen für das STS repräsentieren die vorhandenen Hardwarestrukturen des STS-Detektors und bilden sie mit all ihren individuellen Konfigurationen ab. So gibt es zum Beispiel die Klasse `CbmStsModule`, die die Oberklasse für alle STS Module darstellt und die Klasse `CbmStsSensor`, die die Sensoren repräsentiert. Für das Finden der Cluster und das anschließende Kombinieren der Cluster zu Hits sind zwei individuelle C++-Klassen, `CbmStsFindClusters` und `CbmStsFindHits`, von Dr. Volker Friese entwickelt worden. Beide wurden als Task implementiert, um von der zugrundeliegenden `FairRun` Instanz aufgerufen werden zu können. Durch die Aufteilung der lokalen Rekonstruktion in zwei verschiedene Tasks mussten Ergebnisse des Cluster Finders persistent abgespeichert und für den Hit Finder wieder geladen werden. Ein genereller Ablauf der beiden C++-Klassen bzw. der Funktion dieser ist in Abbildung 3.1, 3.6 und 3.7 gezeigt.
Der erste Schritt dieser Arbeit war das Messen der Zeiten für die beiden Klassen `CbmStsFindClusters` und `CbmStsFindHits`, um so einen Überblick über die Laufzeiten und die Skalierungen der beiden Klassen mit gegebener Anzahl der Events pro Time Slice zu gewinnen. In Abbildung 3.2 sind die Laufzeiten für einen Rechenknoten am FLES mit einem modernen Prozessor, einem Xeon Gold 6130 mit 2.2 GHz, dargestellt. In Abbildung 3.3 sind die Laufzeiten für einen Rechenknoten mit einem älteren Prozessor, einem Xeon Gold E5-2620 mit 2 GHz,

gezeigt. Es wurde zum Vergleich sowohl die eventbasierte Rekonstruktion als auch die zeitbasierte Rekonstruktion für verschiedene Eventanzahlen dargestellt. In der zeitbasierten Rekonstruktion wurden alle Events zu einer Time Slice verarbeitet.
Zu erkennen ist, dass in der eventbasierten Rekonstruktion der Cluster Finder die meiste Zeit verbraucht und linear mit der Anzahl der Events skaliert, während dies in der zeitbasierten Rekonstruktion beim Hit Finder der Fall ist. Die eventbasierte Rekonstruktion skaliert indes linear mit der Anzahl der Events. Im zeitbasierten Fall skaliert die Laufzeit quadratisch mit der Anzahl der Events.

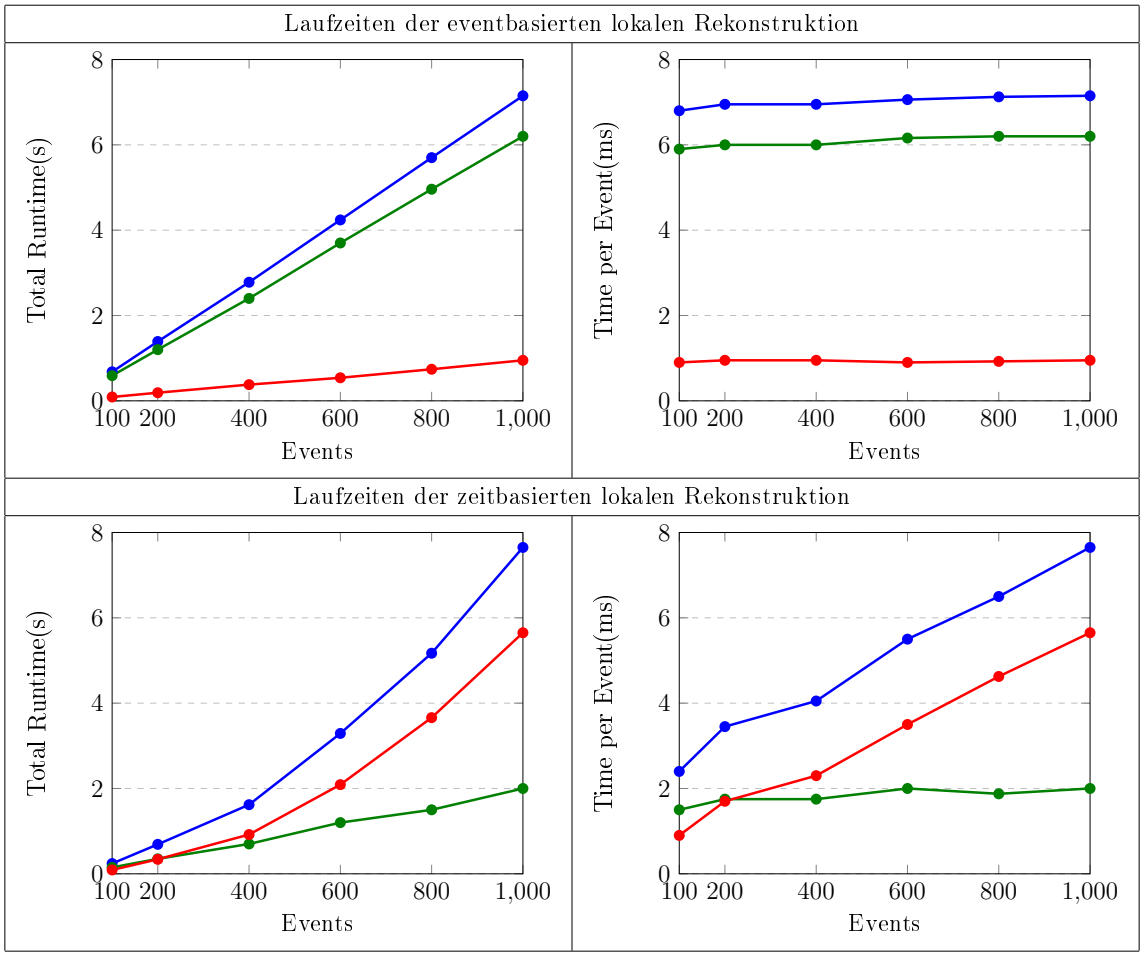


Abbildung 3.2: Laufzeiten der lokalen Rekonstruktion auf einem Intel Xeon Gold 6130 @ 2.2 GHz mit Events aus dem Minimum-Bias Datensatz

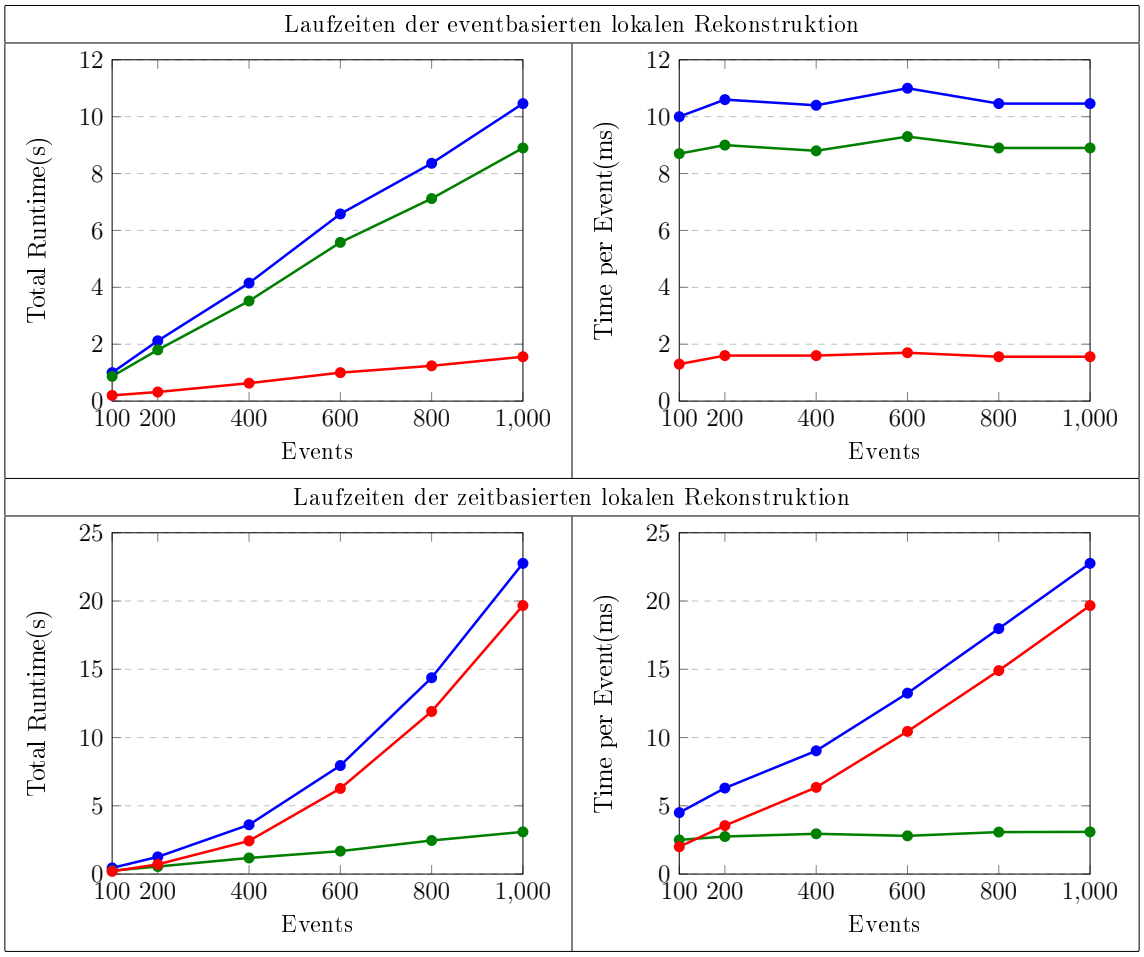


Abbildung 3.3: Laufzeiten der lokalen Rekonstruktion auf einem Intel Xeon E5-2620 @ 2 GHz mit Events aus dem Minimum-Bias Datensatz

3.1 Cluster Finder

Der erste Schritt in der lokalen Rekonstruktion ist das Bilden von Clustern aus den einzelnen Digis der Events oder Time Slices. In beiden Fällen sind die Digis schon nach ihrer Zeit sortiert. Insgesamt ist die Clusterbildung in 5 Schritte aufgeteilt, wobei der letzte Schritt nur für die eventbasierte Rekonstruktion benötigt wird.
Die einzelnen Schritte werden zusammen als Task von der C++-Klasse `CbmStsFindClusters` implementiert und von der Funktion `ProcessData` ausgeführt. Sie sind im Taskablauf in Abbildung 3.6 gezeigt. Zudem ist die Funktion `ProcessData` in Abbildung 7.1 zu sehen. Die 5 Schritte der Clusterbildung sind:

1.) Reset:
In diesem Schritt werden die einzelnen Speicher der Module für die Digis auf ihre Standardwerte gesetzt um die Module zu bereinigen und Fehler auszuschließen.

2.) ProcessDigis:
Die Messdaten aus dem Event oder der Time Slice werden sequentiell den einzelnen Modulen zugeordnet und in diesen weiterverarbeitet. Anhand der zeitlichen und räumlichen Unterschiede der Digis werden diese zu Clustern zusammengefasst.

3.) ProcessBuffer:
Nachdem jedes Digi einem Modul zugeordnet wurde und gleichzeitig Cluster gebildet wurden, bleiben einige Digis im Speicher übrig. Die Speicher werden der Reihe nach abgearbeitet und übrig gebliebene Digis werden zu Clustern verarbeitet.

4.) Analyze:
Die gebildeten Cluster werden genauer analysiert und es werden Clusterparameter berechnet, wie zum Beispiel die Position des Clusters oder auch die durchschnittliche Zeit.

5.) RegisterClusters:
In diesem Schritt werden die Cluster den einzelnen Events zugeordnet, in denen sie gefunden wurden.

Der Algorithmus, der in Schritt 2 und 3 für die Clusterbildung aus den einzelnen Digis benutzt wird, wurde von Dr. Volker Friese entwickelt [12]. Im Prinzip beruht der Algorithmus auf der zeitlichen Vorsortierung der einzelnen Digis. Zwei Digis gehören zum selben Cluster, wenn sie sich in ihren Zeiten nicht mehr als um eine gewisse Grenze unterscheiden und in nebeneinanderliegenden Kanälen gemessen wurden. Ein Cluster wird genau dann erzeugt, sobald in einem schon besetzten oder angrenzenden Kanal ein Digi verarbeitet wird, das zeitlich nicht mehr zu den bereits verarbeiteten Digis passt. Dazu besitzt jedes Modul zwei Speicher, einen Statusspeicher, der die Zeitstempel der Digis enthält, und einen Indexspeicher, der den Index des Digis in der Time Slice oder dem Event an dieser Position enthält. In der folgenden Grafik wird der Algorithmus grafisch dargestellt [12].

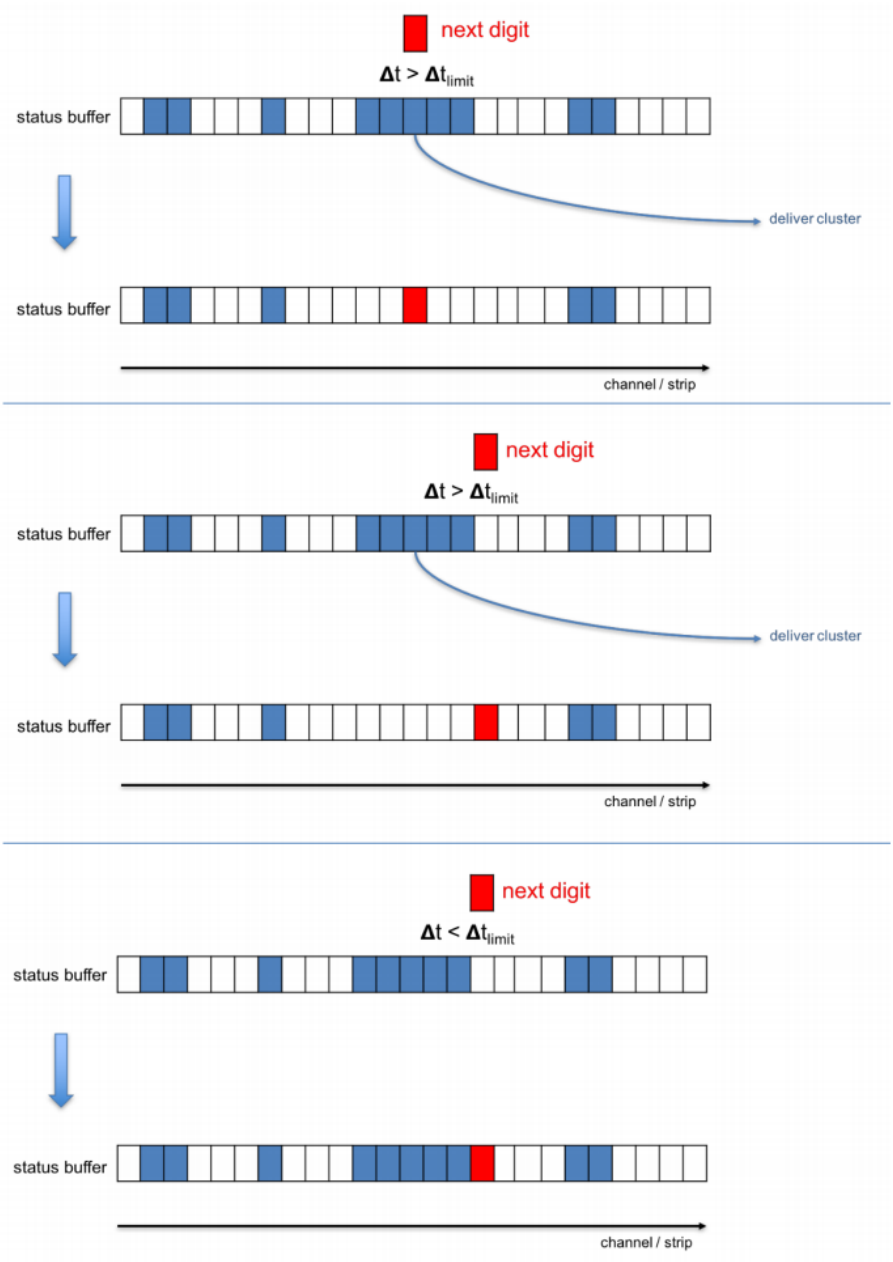


Abbildung 3.4: Grafische Darstellung des Clusterbildungsalgorithmus [12]

Da die Digis nacheinander verarbeitet werden, gibt es 3 verschiedene Fälle, die eintreten können [12]:

1. Ein neues Digi wird in einem leeren Kanal mit leeren Nachbarn eingefügt. Dann kann das Digi in den Statusbuffer mit seiner Zeit eingetragen werden und sein Index wird im Indexbuffer abgelegt.

2. Ein neues Digi soll in einen Kanal eingefügt werden, in dem schon ein Digi vorhanden ist. Dann werden die zusammenhängenden Digis aus den Speichern gelöscht und zu einem Cluster weiterverarbeitet (Abbildung 3.4, oberer Fall).

3. Ein neues Digi soll in einen leeren Kanal eingefügt werden, wobei jedoch mindestens ein Nachbar schon ein Digi enthält. Dann wird der zeitliche Unterschied zwischen dem neu eingefügten und schon vorhandenen Digis geprüft. Falls dieser unter einer bestimmten Schwelle liegt, wird das Digi eingefügt (Abbildung 3.4, unterer Fall). Ansonsten werden alle zusammenhängenden Digis zu einem Cluster gruppiert und aus den Buffern gelöscht (Abbildung 3.4, mittlerer Fall).

Wenn ein Cluster wie im oberen oder mittleren Fall in Abbildung 3.4 gebildet wird, müssen Startpunkt und Endpunkt des Clusters iterativ von der Position des neuen Digi bestimmt werden. Dies geschieht, indem von der Position des neuen Digi in beide Richtung iteriert wird, bis ein leerer Kanal gefunden wurde. Die letzten befüllten Positionen stellen Startpunkt und Endpunkt des Clusters dar. Weiterhin muss, nachdem das letzte Digi aus dem Input verarbeitet wurde, ein letztes Mal über den ganzen Buffer iteriert werden, was Schritt 3 der Clusterbildung entspricht, da nach der Verarbeitung des letzten Digis immer noch Digis im Statusbuffer und Indexbuffer gespeichert sind [12]. Der Speicher muss also geleert werden, um das nächste Event bzw. die nächste Time Slice verarbeiten zu können.

3.2 Hit Finder

Der Hit Finder des STS berechnet aus den vom Cluster Finder erzeugten Clustern Kollisionspunkte mit den Stationen. Dies geschieht in 2 Schritten.
Im ersten Schritt werden die Cluster den Modulen zugeordnet, in denen sie gefunden wurden. Dies entspricht dem Schritt `SortClusters` aus Abbildung 3.6.
Im zweiten Schritt werden je 2 Cluster auf ihre zeitliche und örtliche Nähe untersucht, um schließlich zu einem Hit kombiniert werden zu können. Dafür werden die Cluster der Vorder- oder Rückseite des Moduls zugeordnet. Ein Hit kann immer genau dann auftreten, wenn ein Cluster der Vorderseite und ein Cluster der Rückseite eines Moduls in zeitlicher und räumlicher Nähe liegen. Zuerst wird die zeitliche Nähe überprüft. Es wird jedes Cluster der Vorderseite mit jedem Cluster der Rückseite eines Moduls zeitlich verglichen. Diese Kombinatorik verursacht eine quadratische Laufzeit. Für den zeitlichen Vergleich besitzt jedes Cluster zwei Parameter, eine gemittelte Zeit und einen zeitlichen Fehler, auch Standardabweichung der Zeit oder σ genannt. Beide Parameter wurden vom Cluster Finder aus den Zeiten der Digis des Clusters berechnet. Die quadrierte Standardabweichung der Zeit ergibt die Varianz eines Clusters. Die Varianz stellt gleichzeitig das Zeitintervall dar, welches vom Cluster repräsentiert wird. Folglich sind die Zeitintervalle unterschiedlich groß, da die Standardabweichung und so auch die Varianz für jedes Cluster verschieden sind. Wenn für zwei verglichene Cluster der Betrag der Differenz der gemittelten Zeiten kleiner ist als das Vierfache der Wurzel der addierten Varianzen, könnten beide Cluster einen Hit bilden. Dies entspricht Zeile 35-39 der `FindHits`-Funktion des

originalen Hit Finders, welche in Abbildung 7.3 gezeigt ist. Vereinfacht gesagt überschneiden sich die Zeitintervalle der verglichenen Cluster. Eine Veranschaulichung der Überschneidung von Clustern ist in Abbildung 3.5 dargestellt. Zuletzt werden alle Clusterpaare, die sich in ihren Zeitintervallen überschneiden, auf ihre örtliche Nähe geprüft. Dafür wird die Funktion `IntersectClusters` aufgerufen, welche in Abbildung 7.3 Zeile 39 zu sehen ist.
Schritt 2 enstpricht dem Schritt `FindHitsInModules` aus Abbildung 3.6 und die gesamte Funktion zum Berechnen der Hits ist in Abbildung 7.3 gezeigt und .
Da die vom Cluster Finder erzeugten Cluster nicht nach ihren Zeitintervallen sortiert sind, werden diese in einer zufälligen Reihenfolge verglichen. Aus diesem Grund ist der Hit Output ebenfalls nicht zeitlich sortiert.

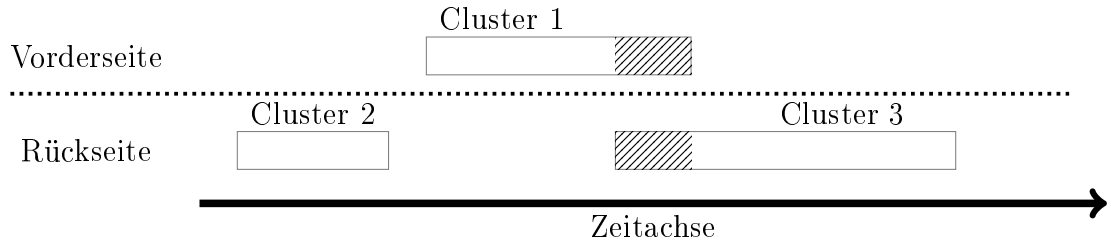
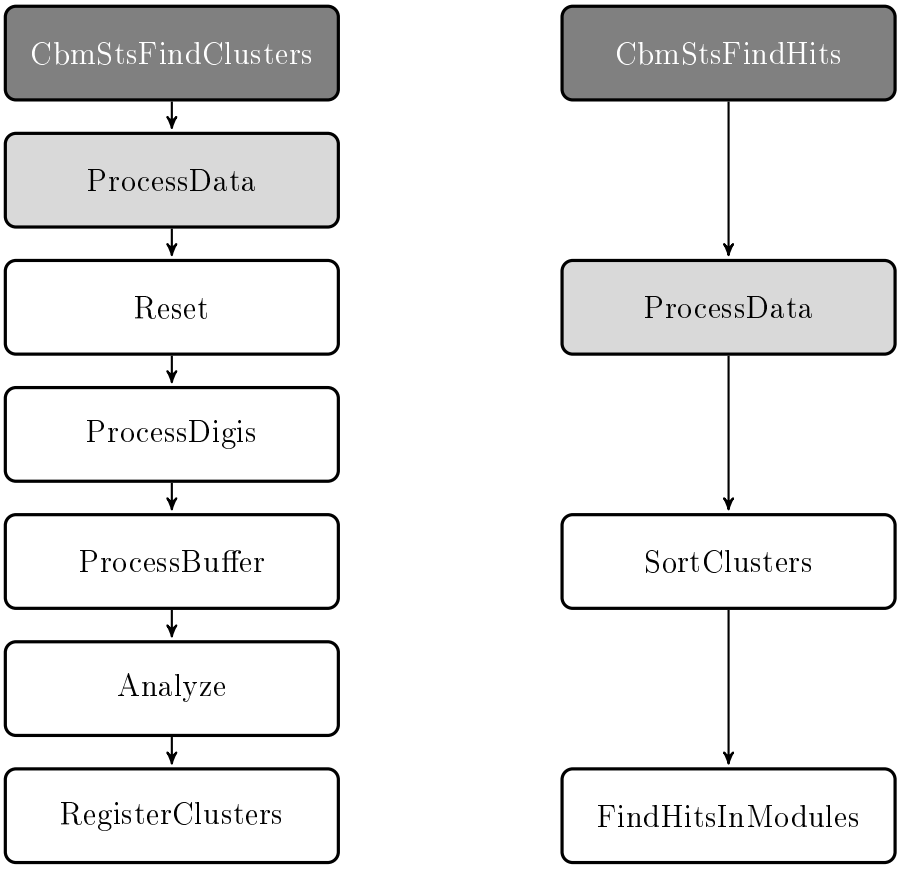


Abbildung 3.5: Zeitliche Überschneidung von zwei Clustern,
Cluster 1 und Cluster 3



(a) Unterschritte von CbmStsFindClusters (b) Unterschritte von CbmStsFindHits

Abbildung 3.6: Tasks `CbmStsFindClusters` und `CbmStsFindHits`

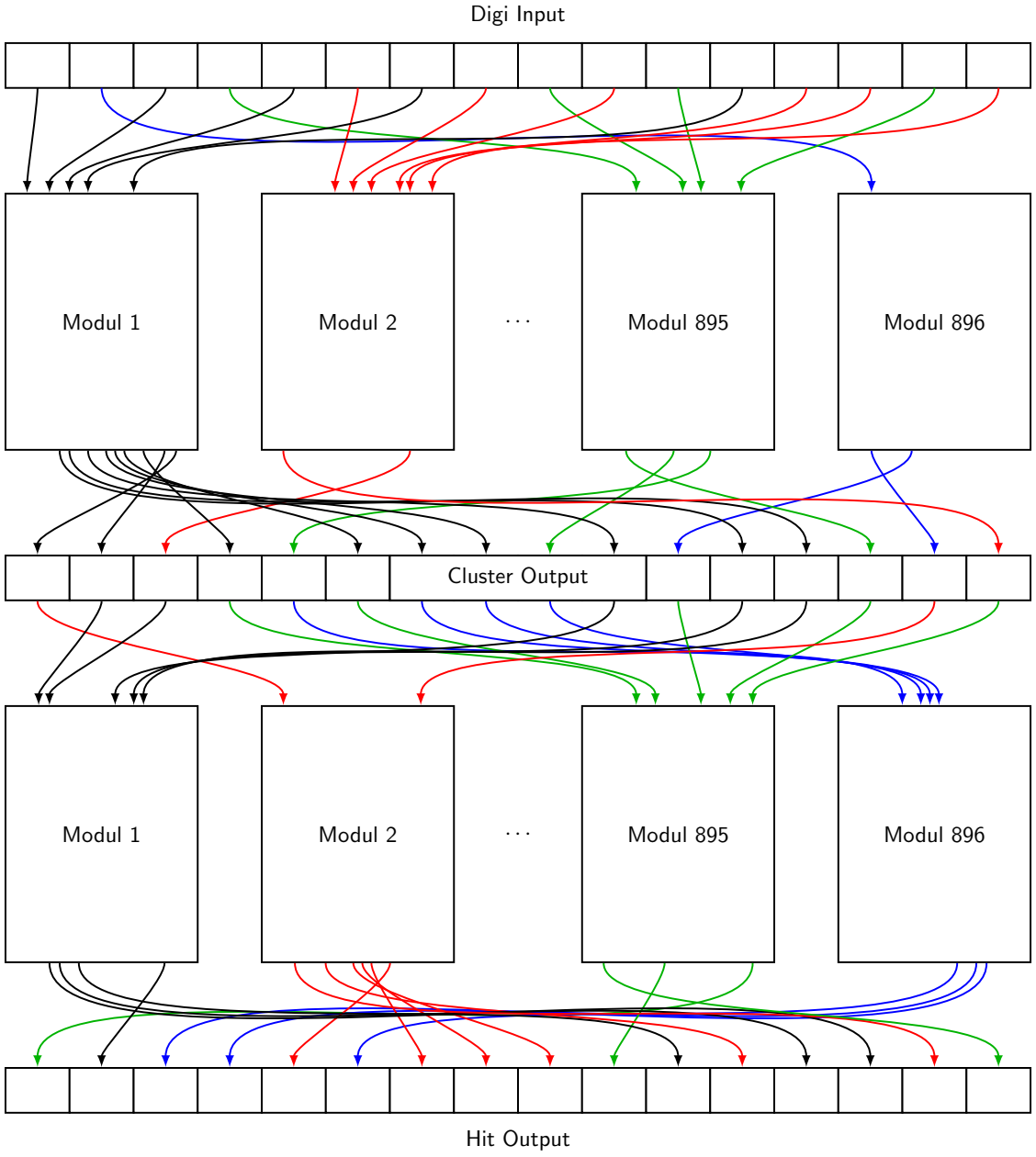


Abbildung 3.7: Grafischer Ablauf der originalen lokalen Rekonstruktion im STS. Zuerst werden die Digis vom Cluster Finder nacheinander in den Modulen zu Clustern verarbeitet und diese abgespeichert. Anschließend lädt der Hit Finder die abgespeicherten Cluster und verarbeitet diese zu Hits, welche wieder gespeichert werden.

4 Optimierung der lokalen Rekonstruktion

Für die Optimierung der lokalen Rekonstruktion in der sequentiellen Ausführung wurden die Zeiten der beiden Tasks `CbmStsFindClusters` und `CbmStsFindHits` gemessen (siehe Kapitel 3 und Abbildung 3.2 und 3.3). Auch die Unterschritte der beiden Tasks (siehe Abbildung 3.6) wurden genauer analysiert, um langsame Programmabschnitte zu identifizieren und zu verbessern.
Für die Optimierung der lokalen Rekonstruktion in der parallelen Ausführung wurden verschiedene Ansätze ausprobiert und die Zeiten dieser verglichen. In den folgenden Abschnitten wird hauptsächlich die aktuell implementierte parallele Version erklären, wobei jedoch teilweise auf die alternativen Möglichkeiten eingegangen wird. Der Fokus wurde auf die zeitbasierte Rekonstruktionskette gelegt, da dies der späteren echten Implementierung und den echten Anforderungen des STS im CBM-Experiment entspricht. Zudem wurden die Time Slices mithilfe des Minimum-Bias Datensatzes generiert.

4.1 Cluster Finder

Beim Cluster Finder wurde überwiegend eine Parallelisierung angestrebt, da der Algorithmus von Dr. Volker Friese bereits speziell für das STS entwickelt und optimiert worden ist [12]. Eine sequentielle Verbesserung im Cluster Finder war das Entfernen des `Reset`-Schrittes. Dieser wird bei einer korrekten Ausführung nicht benötigt, weil `ProcessDigis` und `ProcessBuffer` die Speicher bei einer Clusterbildung bereinigen.
Dr. Volker Friese selbst hat in seiner Zusammenfassung zu dem entwickelten Algorithmus eine parallele Verarbeitung der Digis in allen Modulen als Verbesserung vorgeschlagen [12]. Diese Parallelisierung wurde mit OpenMP angestrebt, brachte allerdings einige Herausforderungen mit sich, da die Digis den einzelenen Modulen sequentiell zugeordnet werden und direkt beim Zuordnen verarbeitet werden. Damit alle Module parallel ihre Digis verarbeiten können, muss jedes Modul alle seine zu verarbeitenden Digis erhalten. Dafür wurden Veränderungen in der C++-Klasse `CbmStsClusterFinderModule` vorgenommen, die die Module des STS repräsentieren. In dieser Klasse bzw. in jedem Modul wurde ein Buffer implementiert, der die Digis aufnimmt. Das Verteilen der Digis hätte nun sequentiell stattfinden können, wie es ursprünglich im originalen Programm der Fall war. Stattdessen wurde eine parallele Verteilung der Digis implementiert, d.h. jeder einzelne Thread/Prozessor-

kern erhält einen Teil der Time Slice und fügt die Digis aus diesem Teil den einzelnen Modulen hinzu. Diese parallele Verteilung ist in Abbildung 7.2 in den Zeilen 10-29 zu sehen. Da dies Schreiboperationen benötigte, wurde jedes Modul mit einer Lock ergänzt, um so die korrekte Ausführung zu gewährleisten. Im späteren CBM-Experiment wird es ca. 900 Module geben und ein Rechenknoten des FLES besitzt maximal 64 Threads. Daher ist Warscheinlichkeit, dass zwei Threads gleichzeitig auf dasselbe Modul zugreifen, gering. Ein Nachteil der parallelen Verarbeitung ist, dass die Digis in den einzelnen Buffern der Module erneut sortiert werden müssen, da die parallele Verteilung die zeitliche Sortierung der Digis in der Time Slice nicht erhält. Hier wurde die in der C++ Standard Library vorhandene `sort`-Funktion verwendet, um die zeitliche Sortierung der Digis für den von Dr. Volker Friese entwickelten Algorithmus wiederherzustellen. Die benötigte Zeit für Sortierung war vernachlässigbar klein.
Zusätzlich zu einem Speicher für die Digis wurde jedes Modul um einen eigenen Speicher für die Cluster erweitert. Grund dafür ist, dass im originalen Programm jedes Modul auf einen gemeinsamen Speicher zugreift, in dem die Cluster abgelegt werden. Für die sequentielle Ausführung stellt dies kein kein Problem dar. Für die parallele Ausführung ist ein gemeinsamer Speicher allerdings ein Problem, denn es könnten Situationen entstehen, in denen mehrere Module gleichzeitig ein Cluster speichern möchten und sich dann gegenseitig überschreiben. Um weiterhin einen gemeinsamen Speicher für die Cluster zu benutzen könnten Konstrukte, wie sie in Abschnitt 2.6 erklärt wurden, verwendet werden. Aufgrund der Häufigkeit einer Clustererstellung, wurde hier stattdessen ein lokaler Buffer für jedes Modul gewählt, um so mögliche entstehende Wartezeiten zu umgehen.
Im optimierten Clusterfinder werden alle Digis parallel ihren Modulen zugeordnet und können anschließend parallel in diesen zu Clustern verarbeitet werden, da es keine Konflikte und Datenabhängigkeiten zwischen den einzelnen Modulen gibt. Dafür verbraucht nun jedes Modul durch die beiden hinzugefügten Buffer mehr Speicherkapazität.
Abbildung 4.2 zeigt den generellen Ablauf für den optimierten Cluster Finder mit den zusätzlichen Speichern der Module. In dieser Abbildung ist auch der optimierte Hit Finder abgebildet, durch den jedes Modul noch einen weiteren Speicher für die aus den Clustern erstellten Hits besitzt. Dies wird im nächsten Kapitel näher erläutert.

4.2 Hit Finder

Im optimierten Hit Finder werden alle Cluster, nachdem diese in jedem Modul gebildet wurden, nach ihrer Zeit sortiert. Die Cluster des Moduls werden daraufhin wie im originalen Hit Finder der Vorder- oder Rückseite des Moduls zugeordnet. Die Cluster

bleiben nach der Zuordnung weiterhin zeitlich sortiert, da sie sequentiell verarbeitet werden. Das hat den Vorteil, dass durch Worst-Case Überlegungen nicht die komplette Kombinatorik durchgeführt werden muss. Im originalen Hit Finder mussten für jedes Clusterpaar von Vorder- und Rückseite die Zeiten und die Fehler der Zeiten ausgelesen und miteinander verrechnet werden. In der optimierten Variante wird direkt beim Zuordnen der Cluster zur Vorder- oder Rückseite der maximale Fehler der Zeiten für beide Seiten bestimmt. Dadurch stehen beim Vergleichen der Clusterpaare zwei Variablen, der maximale Fehler der Zeit für die Vorderseite und der maximale Fehler der Zeit für die Rückseite, zur Verfügung. Aus den beiden Variablen kann weiterhin die maximale zeitliche Differenz für zwei Cluster berechnet werden, welche noch verglichen werden müssen. Die optimierte Version der `FindHits`-Funktion ist in Abbildung 7.4 zu sehen. Die Variable `maxTimeErrorClusterF` stellt den maximalen Fehler für die Cluster der Vorderseite dar, `maxTimeErrorClusterB` analog für die Rückseite. `max_sigma_both` verrechnet beide Variablen zu einem maximalen Gesamtfehler.
Im Folgenden wird die Optimierung mit den Worst-Case-Überlegungen im Hit Finder genauer erklärt und alle Zeilenangaben beziehen sich auf Abbildung 7.4.
Wenn ein Cluster der Vorderseite mit allen Clustern der Rückseite verglichen werden muss, wird ein gemeinsamer Fehler, `max_sigma`, aus dem Fehler des Clusters der Vorderseite mit dem maximalen Fehler der Cluster der Rückseite berechnet, Zeile 50. Beim Vergleichen mit den Clustern der Rückseite kann ein Vergleich abgebrochen werden, sobald die Zeit eines Clusters der Rückseite kleiner ist als die Zeit des Clusters der Vorderseite und die Differenz beider Zeiten außerhalb des Fehlers $4 \cdot$ `max_sigma_both` liegt, Zeilen 60-62. Zudem müssen alle weiteren Cluster der Vorderseite nicht mehr mit diesem Cluster der Rückseite verglichen werden. Dies entspricht den eingesparten Vergleichen im unteren linken roten Dreieck aus Abbildung 4.1. Falls die Zeit des Clusters der Rückseite kleiner ist als die Zeit des Clusters der Vorderseite, die Differenz der Zeiten aber größer als $4 \cdot$ `max_sigma`, so kann der momentane Vergleich abgebrochen werden und es wird das nächste Cluster der Rückseite verglichen, Zeilen 64-65. Wenn die Zeit des Clusters der Rückseite größer ist als die Zeit des Clusters der Vorderseite und die Differenz größer als $4 \cdot$ `max_sigma` ist, muss kein weiteres Cluster der Rückseite mehr mit diesem Cluster der Vorderseite verglichen werden, Zeilen 67-68. Dies entspricht der eingesparten Kombinatorik aus dem oberen rechten roten Dreieck aus Abbildung 4.1. Falls keiner dieser Fälle eintritt, muss auf den Fehler des Clusters der Rückseite zugegriffen werden und wie im originalen Hit Finder die Zeiten und Fehler der Zeiten verglichen werden. Wenn beide Cluster in zeitlicher Nähe liegen, wird schließlich die Funktion `IntersectClusters` aufgerufen, Zeilen 72-78.
Je nach Größe der Fehlerabschätzung im Vergleich zur Größe der Zeitintervalle der Cluster, kann durch diese Überlegungen viel Kombinatorik eingespart werden. In

Abbildung 4.1 ist beispielhaft die übersprungene Kombinatorik des optimierten Hit Finders im Vergleich zur Kombinatorik des originalen Hit Finders veranschaulicht. Beispielweise nehmen wir an, ein Cluster der Vorderseite besitzt einen Zeitpunkt von 900 Zeiteinheiten und `max_sigma_both` beträgt 40 Zeiteinheiten. Sobald dieses Cluster der Vorderseite mit einem Cluster der Rückseite mit 1100 Zeiteinheiten verglichen wird, können wir die Vergleiche für dieses Cluster der Vorderseite abbrechen. Der Betrag der Differenz der beiden Cluster beträgt 200 Zeiteinheiten und ist somit größer als $40 \cdot 4 = 160$ Zeiteinheiten. Alle folgenden Cluster der Rückseite werden einen größeren Betrag der Differenz haben und können somit übersprungen werden, da sie aufsteigend nach ihrer Zeit sortiert sind.
Folglich wird bei guter Fehlerabschätzung die quadratische Laufzeit des Hit Finders für die Clustervergleiche auf eine lineare Laufzeit reduziert.

4.3 DigisToHits

Abschließend wurde eine Vereinigung der beiden Tasks `CbmStsFindClusters` und `CbmStsFindHits` angestrebt. Eine Vereinigung der beiden Tasks bringt einige Vorteile mit sich, beispielsweise wird das Abspeichern und Neuladen der Cluster aus dem Task `CbmStsFindClusters` für den Task `CbmStsFindHits` vermieden. Somit können alle Digis parallel in ihren Modulen zu Hits verarbeitet werden. Dies wird in Abbildung 7.1 in den Zeilen 38-48 gezeigt. Die Schritte `ProcessBuffer` und `Analyze` aus dem Cluster Finder wurden ebenfalls in die Funktion `ProcessDigis` der `CbmStsClusterFinderModule` übernommen. Grund dafür ist, dass die Klasse `CbmStsClusterFinderModule` nun die Cluster speichert. Zudem bietet es sich an, die Parameterberechnungen der Cluster direkt bei der Clustererstellung auszuführen.
Desweiteren ist eine Vereinigung der beiden Tasks sinnvoll, um die Rekonstruktionskette zu vereinfachen und eine klare Struktur der beiden Tasks vorzugeben: Es können keine Hits gefunden werden, wenn zuvor keine Cluster gebildet worden sind.

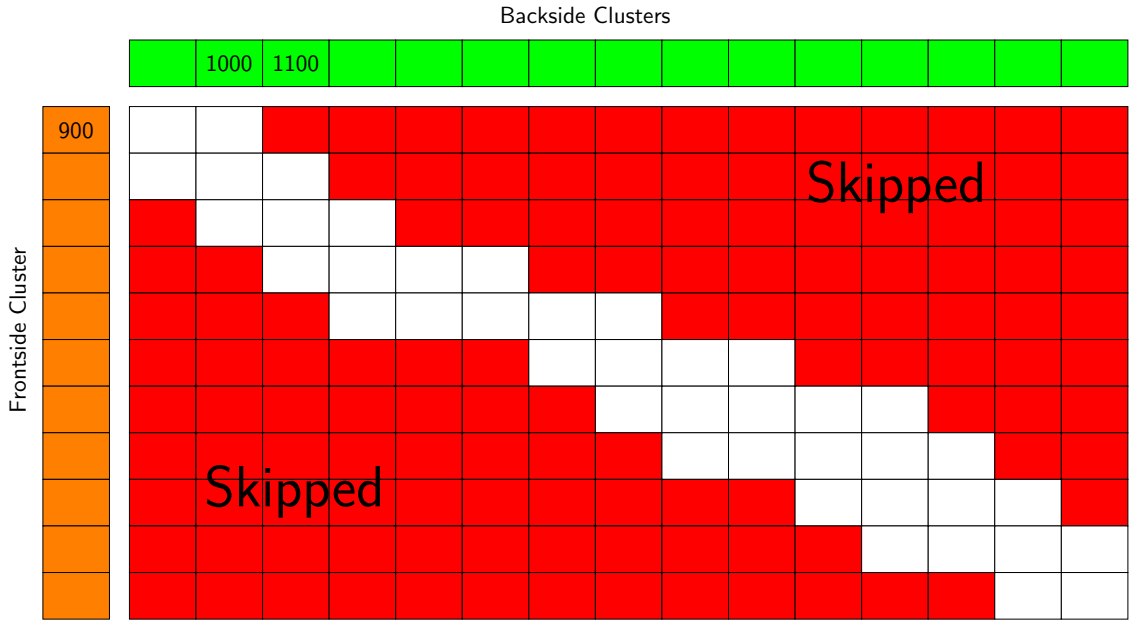


Abbildung 4.1: Reduzierte Kombinatorik im optimierten Hit Finder

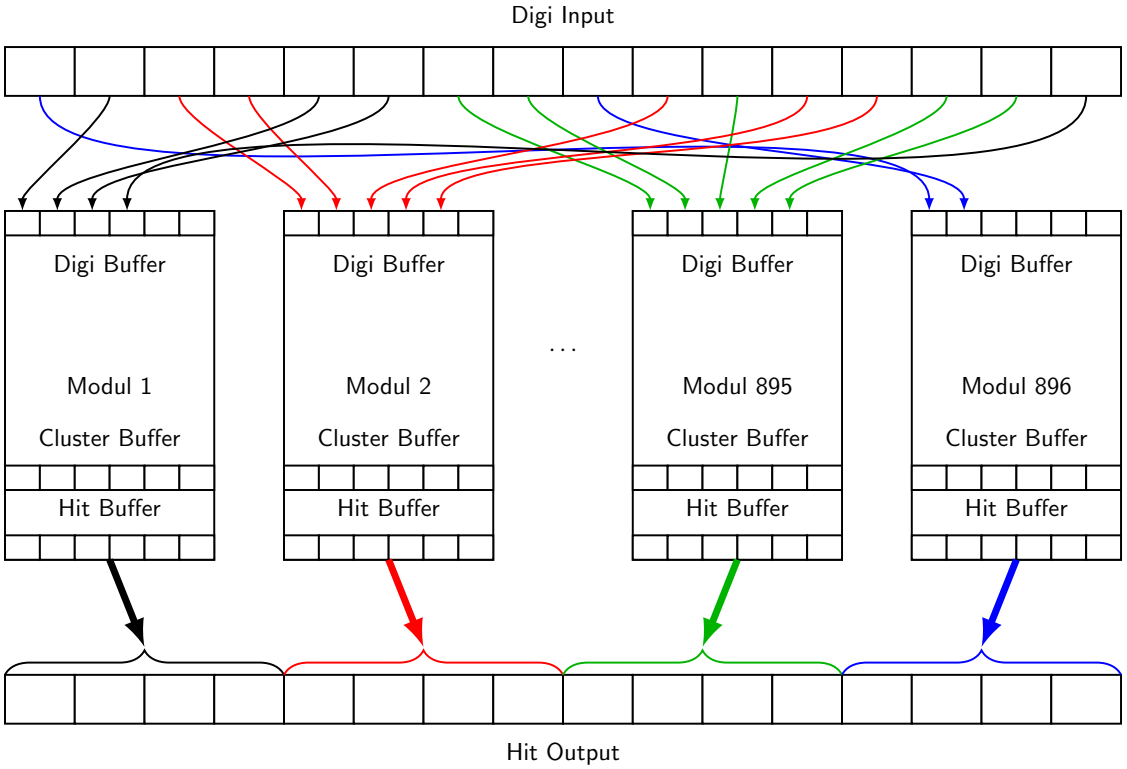


Abbildung 4.2: Grafischer Ablauf der optimierten lokalen Rekonstruktion im STS

5 Ergebnisse

Für das Testen und zur Veranschaulichung der Optimierung wurde die Rekonstruktion im zeitbasierten Modus betrachtet. Außerdem wurde der Minimum-Bias Datensatz gewählt, der die realitätsnahen Daten mit Messrauschen enthält, die vom CBM-Experiment voraussichtlich gemessen werden. Aus diesem Datensatz wurden Time Slices simuliert, denen verschiedene Anzahlen an Kollisionsevents zu Grunde liegen. Es wurden zwischen 100 und 1000 Kollisionsereignisse gewählt da dies ca. der Zielgröße einer Time Slice entspricht, wie sie später vom FLES verarbeitet werden soll [14].
Für das Testen der Skalierfähigkeit des Programms mit Kernen/Threads wurde als Datensatz eine Time Slice mit 1000 zugrundeliegenden Events benutzt, um eine große Eingabemenge von ca. 5 Millionen Digis zu erhalten.
Alle simulierten Kollisionsevents wurden zu genau einer Time Slice verarbeitet, da die Erweiterung von der Ausführung einer Time Slice auf mehrere Time Slices trivial ist. Die benötigte Zeit für das Löschen der Speicher und das Zurücksetzen der Module des Tasks `CbmStsDigisToHits` ist vernachlässigbar klein.
Durch die Parallelisierung des neuen Task `CbmStsDigisToHits` und implementierte Umstrukturierungen, sind der Cluster Finder und der Hit Finder laufzeittechnisch nicht mehr voneinander trennbar. Folglich konnten keine einzelnen Vergleiche zwischen originalem und optimiertem Cluster Finder erstellt werden. Für den Hit Finder war dies noch möglich, da die Verbesserungen modular sind und sich nur in einigen Funktionen der Unterklassen finden. Ein Vergleich zwischen dem originalen und optimierten Hit Finder ist in Abbildung 5.1 zu sehen.
Die sequentielle Ausführung des neuen Tasks `CbmStsDigisToHits` für die lokale Rekonstruktion im Vergleich zu den vorherigen beiden Tasks ist in Abbildung 5.2 gezeigt. Zu erkennen ist, dass die Laufzeit drastisch gesunken ist und nun linear, statt quadratisch, mit der Eventanzahl skaliert. Dies wird vor allem deutlich, wenn die rechten Graphen betrachtet werden, in denen die Laufzeit pro Event aufgetragen ist.
Alle Vergleiche wurden für 2 verschiedene Rechenknoten des FLES mit unterschiedlichen Prozessoren durchgeführt, da sich die Laufzeiten und Ergebnisse für diese teilweise stark unterscheiden. Verwendet wurde ein Rechenknoten mit einem Intel Xeon Gold 6130 mit 2.2 GHz, welcher einen relativ modernen Prozessor darstellt. Ein zweiter Rechenknoten mit einem Intel Xeon E5-2620 mit 2 GHz wurde ebenfalls verwendet. Obwohl sich beide Prozessoren in der Taktrate relativ ähnlich sind,

waren in der sequentiellen Ausführung deutliche Zeitunterschiede erkennbar. Sowohl für den Hit Finder, als auch für die gesamte lokale Rekonstruktion waren die Unterschiede in der sequentiellen Laufzeit enorm, siehe dazu Abbildung 5.1 und 5.2.
Der Speedup der neuen lokalen Rekonstruktion im Vergleich zur alten Rekonstruktion für beide Prozessoren ist in Abbildung 5.2, unten, gezeigt. Zu erkennen ist, dass der Speedup für Time Slices mit mehr zugrundeliegenden Events größer ist als für Time Slices mit weniger zugrundeliegenden Events. Das liegt vor allem an der Verbesserung des Hit Finders, wo mehr Kombinatorik für größere Eventanzahlen eingespart wurde und somit eine deutlich bessere Laufzeit erzielt wurde. Für die vorgeschlagene Größe von 1000 Events pro Time Slice wurde somit der größte Speedup erreicht [14].

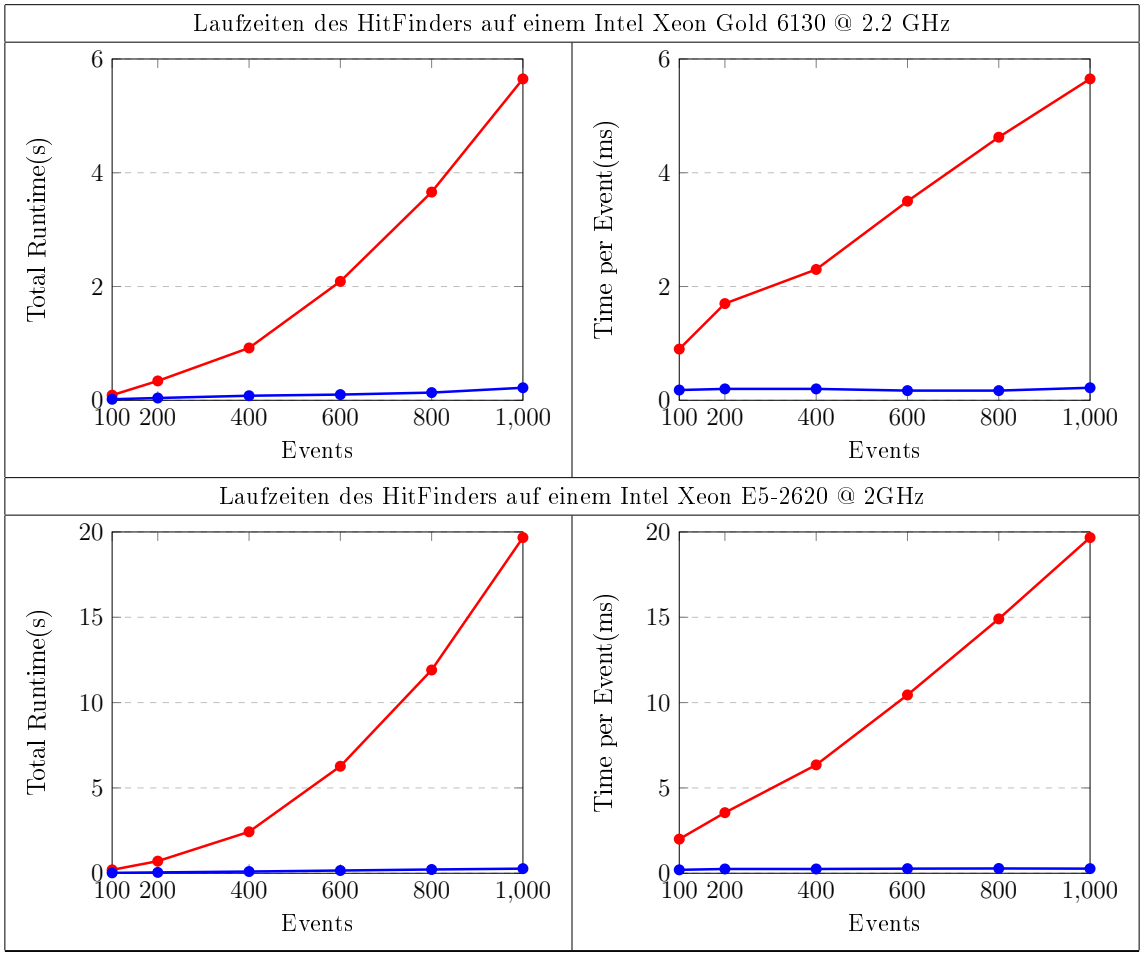


Abbildung 5.1: Laufzeiten des originalen und optimierten HitFinders für verschiedene Größen der Time Slices auf verschiedenen CPUs des FLES mit Events aus dem Minimum-Bias Datensatz

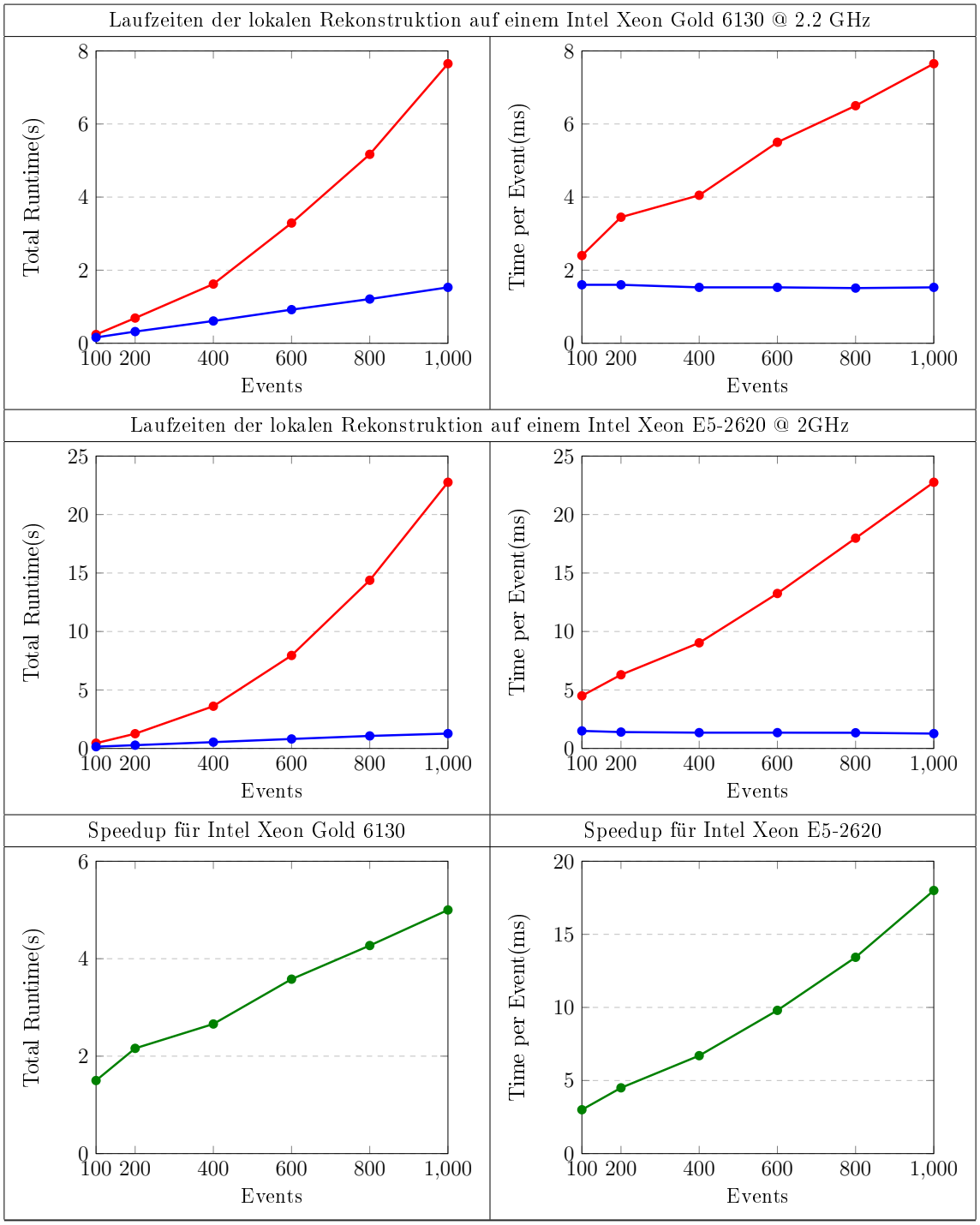


Abbildung 5.2: Laufzeiten der originalen und optimierten lokalen Rekonstruktion für verschiedene Größen der Time Slices auf verschiedenen CPUs des FLES sowie der daraus resultierte Speedup nach dem Ahmdalschen Gesetz (Gleichung 2.1) mit Events aus dem Minimum-Bias Datensatz

Ein Rechenknoten des FLES besitzt immer 2 Prozessoren gleicher Art, sodass für einen Knoten mit 2 Intel Xeon Gold 6130 mit je 16 Kernen bzw. 32 Threads pro Prozessor insgesamt 32 Kerne bzw. 64 Threads vorhanden sind. Für einen Knoten mit 2 Intel Xeon E5-2620 gibt es insgesamt 12 Kerne bzw. 24 Threads.
Die Skalierung des Tasks `CbmStsDigisToHits` wurde daher für beide Knoten mit verschiedenen Anzahlen der Kerne/Threads getestet und ist in Abbildung 5.3 dargestellt. Auffällig ist, dass die Skalierung anfangs linear mit der Anzahl der Kerne/-Threads verläuft, im späteren Verlauf steigt die Laufzeit. Eine Erklärung für den Anstieg bei höherer Anzahl der Kerne/Threads ist das Erreichen des Maximums der physikalischen Rechenkerne. Somit wird nur noch eine virtuelle Parallelisierung mit den Threads erzeugt, welche aber zu keinem Speedup mehr führen kann. Bei dem Intel Xeon Gold 6130 ist dieses Maximum bei 32 Kernen erreicht, beim Intel Xeon E5-2620 ist dies bei 12 Kernen der Fall. Dennoch ist zu sehen, dass beim Intel Xeon Gold 6130 bereits vor dem Ausnutzen der maximalen physikalischen Kerne eine Verlangsamung der Gesamtlaufzeit eintritt. Diese Verlangsamung wurde in dieser Arbeit nicht weiter analysiert. Mögliche Gründe könnten sein, dass der schnelle Speicher der CPU für viele Kerne nicht ausreicht oder die Reduktion der Ergebnisse im Task mit OpenMP ineffizient abläuft. Ein weiterer Grund könnte sein, dass durch zu viele Kerne/Threads die Arbeit nicht gleichmäßig verteilt wird, da eine parallele Verarbeitung der STS Module implementiert wurde. Es kollidieren weitaus mehr geladene Teilchen mit den STS Modulen, die in der Mitte einer STS Station platziert sind, als mit weiter außen platzierten Modulen [2]. Das heißt, dass die Digis innerhalb einer Time Slice nicht gleichmäßig auf alle Module verteilt werden. Somit könnte es sein, dass eine Parallelisierung auf Modulebene weniger sinnvoll ist als angenommen.
Beim Intel Xeon E5-2620, ist erkennbar, dass die Laufzeit bis zum physikalischen Limit von 12 Kernen gut skaliert. Ab dem Erreichen des maximalen physikalischen Kernanzahl steigt die Laufzeit leicht an.
Anzumerken ist, dass in beiden Fällen bei einer Ausführung auf nur einem Kern die Laufzeiten wesentlich langsamer sind als in der sequentiellen Variante, in der OpenMP nicht benutzt wird. Zum Beispiel beträgt die sequentielle Laufzeit für einen Intel Xeon Gold 6130 ca. 1,5 Sekunden, für die parallele Ausführung mit einem Thread, `OMP_Num_Threads = 1`, beträgt die Laufzeit dagegen ca. 4,4 Sekunden. Für Laufzeitmessungen, welche im Graphen verwendet werden, wurde festgelegt, wie viele Kerne für die lokale Rekonstruktion benutzt werden sollen und eine dynamische Threadanzahl mit `OMP_Dynamic = false` verhindert. Möglicherweise werden bei der sequentiellen Ausführung auf einer CPU durch den Prozessor selbst mehrere Kerne/Threads teilweise verwendet oder der durch OpenMP produzierte Overhead trägt zu einer erheblichen Verlangsamung auf nur einem Kern bei.

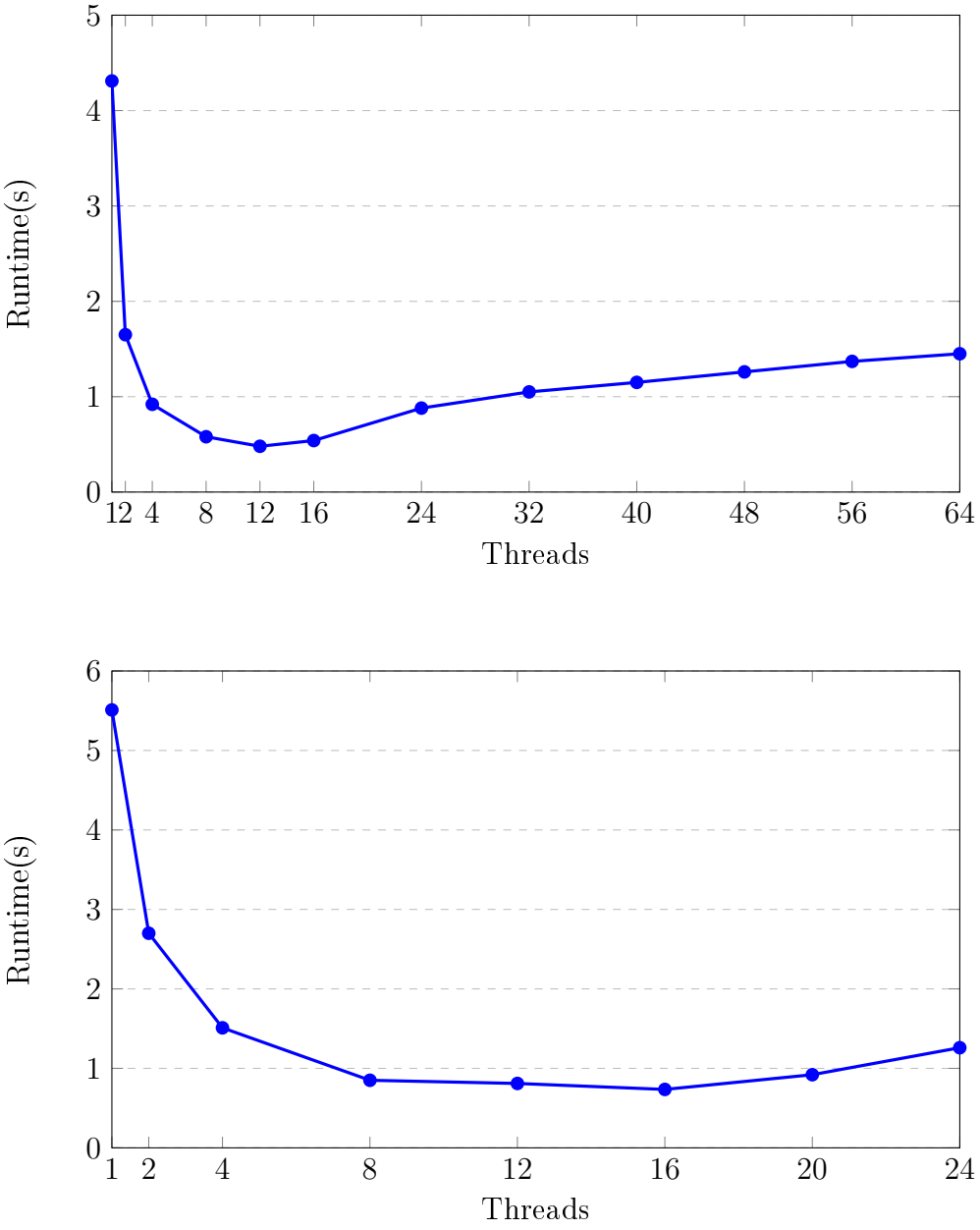


Abbildung 5.3: Skalierung des Tasks `CbmStsDigisToHits` anhand einer Time Slice mit 1000 zugrundeliegenden Kollisionsereignissen aus dem Minimum-Bias Datensatz
Oben: Intel Xeon Gold 6130 mit 32 physikalischen Kernen
Unten: Intel Xeon E5-2620 mit 12 physikalischen Kernen

6 Zusammenfassung

Die Optimierung des Cluster Finders hat einzelne Schritte des Tasks `CbmStsFindClusters` zusammengefasst und umstrukturiert. Dies führte jedoch zu keiner wesentlichen Verbesserung der Laufzeit.
Die Optimierung des Hit Finders hat eine beachtliche Verbesserung der Laufzeit des Hit Finders erzielt und somit auch die gesamte Laufzeit der lokalen Rekonstruktion im zeitbasierten Modus deutlich verbessert .
Das anschließende Zusammenfügen der beiden Tasks zum Task `CbmStsDigisToHits` hat den Vorteil, dass die Cluster nicht mehr persistent abgespeichert werden müssen und einige Berechnungen direkt beim Erstellen der Cluster ausgeführt werden können. Dadurch wurden Datenzugriffe effizienter gestaltet und insgesamt weniger Datenzugriffe benötigt. Außerdem müssen nicht mehr die Parameter, die in beiden Tasks `CbmStsFindClusters` und `CbmStsFindHits` benutzt wurden, doppelt angegeben werden, wodurch die Redundanz verringert werden konnte.
Insgesamt wurde in der sequentiellen Ausführung ein bis zu 18-facher Speedup der lokalen Rekonstruktion erreicht, der mit der Anzahl der zugrundeliegenden Events pro Time Slice skaliert.
Eine Parallelisierung des Tasks `CbmStsDigisToHits` auf Modulebene wurde für eine Time Slice ermöglicht und eine gute Skalierung der Laufzeit für die Rechenknoten des FLES mit den Intel Xeon E5-2620 CPUs erreicht. Für die Skalierung der Laufzeit auf Rechenknoten mit den Intel Xeon Gold 6130 CPUs sollten weitere Analysen durchgeführt werden. Die Analysen sollten die Schwachstellen identifizieren, die die Laufzeit beeinträchten und eine bessere Skalierung verhindern. Ebenso bleibt offen, warum der Overhead mit der Benutzung von OpenMP die Laufzeit des Tasks für einen einzelnen Kern/Thread beinahe vervierfacht. Auch weitere Tests mit anderen Größen der Time Slices könnten mehr Einblicke in die Skalierung des Tasks `CbmStsDigisToHits` bringen. Auf dem jetzigen Stand bietet die implementierte Parallelisierung einen guten Ansatz, der noch weiter vertieft werden sollte, wenn eine performantere Parallelisierung auf Modulebene angestrebt wird.

7 Anhang

Abbildung 7.1: `ProcessData`-Funktion bzw. 5 Schritte des originalen Cluster Finders

```
1  // --- Reset all cluster finder modules
2  fTimer.Start();
3  for (auto it = fModules.begin(); it != fModules.end(); it++)
4          it->second->Reset();
5  fTimer.Stop();
6  Double_t time1 = fTimer.RealTime();
7
8  // --- Start index of newly created clusters
9  Int_t indexFirst = fClusters->GetEntriesFast();
10
11 // --- Number of input digis
12 fTimer.Start();
13 Int_t nDigis = (event ? event->GetNofData(kStsDigi): fDigis->
       GetEntriesFast() );
14
15 // --- Loop over input digis
16 Int_t digiIndex = -1;
17 for (Int_t iDigi = 0; iDigi < nDigis; iDigi++) {
18         digiIndex = (event ? event->GetIndex(kStsDigi, iDigi) : iDigi);
19         ProcessDigi(digiIndex);
20 }  //# digis in time slice or event
21 fTimer.Stop();
22 Double_t time2 = fTimer.RealTime();
23
24 // --- Process remaining clusters in the buffers
25 fTimer.Start();
26 for (auto it = fModules.begin(); it != fModules.end(); it++)
27         it->second->ProcessBuffer();
28 fTimer.Stop();
29 Double_t time3 = fTimer.RealTime();
30
31 // --- Stop index of newly created clusters
32 Int_t indexLast = fClusters->GetEntriesFast();
33
34 // --- Determine cluster parameters
35 fTimer.Start();
36 for (Int_t index = indexFirst; index < indexLast; index++) {
```

```
37          CbmStsCluster* cluster = dynamic_cast<CbmStsCluster*>(fClusters
                ->At(index));
38          CbmStsModule* module = dynamic_cast<CbmStsModule*>
39          (fSetup->GetElement(cluster->GetAddress(), kStsModule));
40          fAna->Analyze(cluster, module, fDigis);
41  }
42  fTimer.Stop();
43  Double_t time4 = fTimer.RealTime();
44
45  // --- In event-by-event mode: register clusters to event
46  fTimer.Start();
47  if ( event ) {
48          for (Int_t index = indexFirst; index < indexLast; index++)
49                  event->AddData(kStsCluster, index);
50  } //? Event object
51
52  fTimer.Stop();
53  Double_t time5 = fTimer.RealTime();
```

Abbildung 7.2: `ProcessData`-Funktion des Tasks `CbmStsDigisToHits`

```
1
2   // --- Start index of newly created clusters
3   Int_t indexFirst = fClusters->GetEntriesFast();
4
5   // --- Number of input digis
6   fTimer.Start();
7   Int_t nDigis = (event ? event->GetNofData(kStsDigi): fDigis->
        GetEntriesFast() );
8
9   // --- Loop over input digis and assign each digi into its module,
        mixing the order of the digis
10  Int_t digiIndex = -1;
11  #pragma omp parallel for firstprivate(digiIndex) schedule(static) if(
        parallelism_enabled)
12  for (Int_t iDigi = 0; iDigi < nDigis; iDigi++){
13
14          digiIndex = (event ? event->GetIndex(kStsDigi, iDigi) : iDigi);
15
16          CbmStsDigi* digi = dynamic_cast<CbmStsDigi*> (fDigis->At(
                digiIndex));
17          assert(digi);
18          digi->SetIndex(digiIndex);
19
20          CbmStsClusterFinderModule* module = fModules.at(digi->
                GetAddress());
21          assert(module);
22
23          // --- Digi channel
24          UShort_t channel = digi->GetChannel();
25          assert ( channel < module->GetSize() );
26
27          // add digi to digi buffer of module
28          module->AddDigiToQueue(digi);
29  }
30
31  fTimer.Stop();
32  Double_t time2 = fTimer.RealTime();
33  time2_total += time2;
34
35  // --- Process each module, sort digis by time, processdigis,
        processbuffers, analyse and assign clusters to modules
36  fTimer.Start();
37
38  // Reduction to collect all Hits
39  #pragma omp declare reduction(combineOutput:TClonesArray*: omp_out->
        AbsorbObjects(omp_in)) initializer(omp_priv = new TClonesArray("
```

```
    CbmStsHit", 1e1))
40
41  #pragma omp parallel for schedule(static) reduction(combineOutput:fHits
        ) if(parallelism_enabled)
42  for (Int_t it = 0; it < fModules.size(); it++){
43
44          // find hits in module and reduce them into fHits
45          fHits->AbsorbObjects(fModuleIndex[it]->ProcessDigis());
46          //LOG(INFO) << "Module " << it << " finished " << FairLogger::
                endl;
47
48  }
49  fTimer.Stop();
50  Double_t time3 = fTimer.RealTime();
51  time3_total += time3;
52
53  // --- Stop index of newly created clusters
54  Int_t indexLast = fClusters->GetEntriesFast();
55
56  // --- In event-by-event mode: register clusters to event
57  fTimer.Start();
58  if ( event ) {
59          for (Int_t index = indexFirst; index < indexLast; index++)
60                  event->AddData(kStsCluster, index);
61  } //? Event object
62
63  fTimer.Stop();
64  Double_t time5 = fTimer.RealTime();
65
66  fHits->Delete();
```

Abbildung 7.3: `FindHits`-Funktion des originalen Hit Finders

```
1  Int_t nHits = 0;
2
3  Int_t nClusters = clusters.size();
4
5  Int_t nClustersF = 0;
6  Int_t nClustersB = 0;
7
8  // --- Sort clusters into front and back side
9  vector<Int_t> frontClusters;
10 vector<Int_t> backClusters;
11 Int_t side  = -1;          // front or back side
12 for (Int_t iCluster = 0; iCluster < nClusters; iCluster++) {
13         CbmStsCluster* cluster = clusters[iCluster];
14         side = GetSide( cluster->GetPosition() );
15
16         if ( side == 0) {
17                 frontClusters.push_back(iCluster);
18                 nClustersF++;
19         }
20         else if ( side == 1 ) {
21                 backClusters.push_back(iCluster);
22                 nClustersB++;
23         }
24         else
25                 LOG(FATAL) << GetName() << ": Illegal side qualifier "
                       << side << FairLogger::endl;
26 }  // Loop over clusters in module
27
28
29 // --- Loop over front and back side clusters
30 for (Int_t iClusterF = 0; iClusterF < nClustersF; iClusterF++) {
31         CbmStsCluster* clusterF = clusters[frontClusters[iClusterF]];
32         for (Int_t iClusterB = 0; iClusterB < nClustersB;   iClusterB
              ++) {
33                 CbmStsCluster* clusterB = clusters[backClusters[
                       iClusterB]];
34
35                 Double_t sigma = TMath::Sqrt( clusterF->GetTimeError()
                       * clusterF->GetTimeError() + clusterB->GetTimeError
                       () * clusterB->GetTimeError() );
36                 if ( fabs(clusterF->GetTime() - clusterB->GetTime()) >
                       4. * sigma ) continue;
37
38                 // --- Calculate intersection points
39                 Int_t nOfHits = IntersectClusters(clusterF, clusterB);
40                 nHits += nOfHits;
```

```
41
42          }  // back side clusters
43
44  }  // front side clusters
45
46  return nHits;
```

Abbildung 7.4: `FindHits`-Funktion des optimierten Hit Finders

```
1  Int_t nHits = 0;
2
3  Int_t nClusters = clusters.size();
4
5  Int_t nClustersF = 0;
6  Int_t nClustersB = 0;
7
8  // --- Sort clusters into front and back side
9  vector<Int_t> frontClusters;
10 vector<Int_t> backClusters;
11 // Max Time Errors for Frontside and Backside Clusters
12 Double_t maxTimeErrorClustersF = 0.;
13 Double_t maxTimeErrorClustersB = 0.;
14 Int_t side  = -1;          // front or back side
15 for (Int_t iCluster = 0; iCluster < nClusters; iCluster++) {
16         CbmStsCluster* cluster = clusters[iCluster];
17         side = GetSide( cluster->GetPosition() );
18
19         if ( side == 0) {
20                 frontClusters.push_back(iCluster);
21                 nClustersF++;
22                 // Find maximum error
23                 if ( cluster->GetTimeError() > maxTimeErrorClustersF )
                       maxTimeErrorClustersF = cluster->GetTimeError();
24         }
25         else if ( side == 1 ) {
26                 backClusters.push_back(iCluster);
27                 nClustersB++;
28                 // Find maximum error
29                 if ( cluster->GetTimeError() > maxTimeErrorClustersB )
                       maxTimeErrorClustersB = cluster->GetTimeError();
30         }
31         else
32                 LOG(FATAL) << GetName() << ": Illegal side qualifier "
                       << side << FairLogger::endl;
33 }  // Loop over clusters in module
34
35 // --- Loop over front and back side clusters
36 // Starting point of the comparisons in each iteration
37 Int_t startB = 0;
38
39 const Double_t max_sigma_both = TMath::Sqrt(maxTimeErrorClustersF *
       maxTimeErrorClustersF + maxTimeErrorClustersB *
       maxTimeErrorClustersB);
40
41
```

```
42  for (Int_t iClusterF = 0; iClusterF < nClustersF; iClusterF++) {
43          CbmStsCluster* clusterF = clusters[frontClusters[iClusterF]];
44
45          Double_t clusterF_Time = clusterF->GetTime();
46
47          Double_t clusterF_TimeError = clusterF->GetTimeError() *
                clusterF->GetTimeError();
48
49          // Maximum difference between the two times of the current
                cluster and the maximum of the backside cluster
50          Double_t max_sigma = TMath::Sqrt( clusterF_TimeError +
                maxTimeErrorClustersB * maxTimeErrorClustersB );
51
52          for (Int_t iClusterB = startB; iClusterB < nClustersB;
                iClusterB++) {
53                  CbmStsCluster* clusterB = clusters[backClusters[
                        iClusterB]];
54
55
56                  Double_t timeDiff = clusterF_Time - clusterB->GetTime()
                        ;
57
58                  // If Time Difference is greater than the maximum error
                        , we can shift the beginning to the right
59                  // If the difference is bigger than the max sigma (
                        front cluster, max back cluster) we can start the
                        next iteration
60                  if ( ( timeDiff > 0 ) && ( timeDiff > 4. *
                        max_sigma_both)){
61                          startB++;
62                          continue;
63                  }
64                  else if ( ( timeDiff > 0 ) && ( timeDiff > 4. *
                        max_sigma ) ) {
65                          continue;
66                  }
67                  else if ( ( timeDiff < 0 ) && ( fabs(timeDiff) > 4. *
                         max_sigma ) ) {
68                          break;
69                  }
70                  else {
71
72                          Double_t sigma = TMath::Sqrt(
                                clusterF_TimeError + clusterB->GetTimeError
                                () * clusterB->GetTimeError() );
73                          if ( fabs(timeDiff) > 4. * sigma ) {
74                                  continue;
75                          }
```

```
76
77                          // --- Calculate intersection points
78                          Int_t nOfHits = IntersectClusters(clusterF,
                                clusterB);
79
80          } // back side clusters
81
82  } // front side clusters
83
84  return nHits;
```

Literaturverzeichnis

[1] Compressed Baryonic Matter. https://www.gsi.de/work/forschung/cbmnqm/cbm.htm. Zuletzt besucht am: 6. Juni 2019.

[2] GSI. Technical Desgin Report for the CBM. http://repository.gsi.de/record/54798/files/GSI-Report-2013-4.pdf. Zuletzt besucht am: 6. Juni 2019.

[3] Hanna Malygina. Hit reconstruction for the Silicon Tracking System of the CBM experiment. https://indico.gsi.de/event/7235/contribution/0/material/0/0.pdf. Zuletzt besucht am: 6. Juni 2019.

[4] Mélissa Gaillard. CERN Data Centre passes the 200-petabyte milestone. https://home.cern/news/news/computing/cern-data-centre-passes-200-petabyte-milestone. Zuletzt besucht am: 6. Juni 2019.

[5] Storage - What data to record? https://home.cern/science/computing/storage. Zuletzt besucht am: 6. Juni 2019.

[6] LHC Datenverarbeitung. http://www.lhc-facts.ch/index.php?page=datenverarbeitung. Zuletzt besucht am: 6. Juni 2019.

[7] Green IT Cube. https://ttsp-hwp.de/de/projects/green-it-cube/. Zuletzt besucht am: 6. Juni 2019.

[8] CPU-Grundlagen: Multithreading. https://www.tecchannel.de/a/cpu-grundlagen-multithreading,402289. Zuletzt besucht am: 6. Juni 2019.

[9] intel. Intel Streaming SIMD Extensions Technology. https://www.intel.de/content/www/de/de/support/articles/000005779/processors.html. Zuletzt besucht am: 6. Juni 2019.

[10] Facility for Antiproton and Ion Research in Europe. https://fair-center.de/. Zuletzt besucht am: 6. Juni 2019.

[11] FAIR Beschleunigeranlage. https://fair-center.de/de/oeffentlichkeit/was-ist-fair.html. Zuletzt besucht am: 6. Juni 2019.

[12] Volker Friese. A cluster-finding algorithm for free-streaming data. https://indico.gsi.de/event/7440/contribution/6/material/poster/0.pdf. Zuletzt besucht am: 6. Juni 2019.

[13] Johann M. Heuser. Status of the CBM experiment. http://inspirehep.net/record/1374298/files/epjconf_icnfp2014_01006.pdf. Zuletzt besucht am: 6. Juni 2019.

[14] J de Cuveland and V Lindenstruth and. A first-level event selector for the CBM experiment at FAIR. *Journal of Physics: Conference Series*, 331(2):022006, dec 2011. Zuletzt besucht am: 6. Juni 2019.

[15] FairRoot. https://cbmroot.gsi.de/?q=node/1. Zuletzt besucht am: 6. Juni 2019.

[16] CbmRoot. http://web-docs.gsi.de/~uhlig/doxygen/html/index.html. Zuletzt besucht am: 6. Juni 2019.

[17] CERN. ROOT. https://root.cern.ch/. Zuletzt besucht am: 6. Juni 2019.

[18] FairRootGroup. FairRoot Github. https://github.com/FairRootGroup/FairRoot.

[19] Florian Uhlig Mohammad Al-Turany. FairRoot Framework. https://pos.sissa.it/070/048/pdf. Zuletzt besucht am: 6. Juni 2019.

[20] OpenMP Specifications. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. Zuletzt besucht am: 6. Juni 2019.

[21] heise.de. Bösartige Race Conditions und Data Races. https://www.heise.de/developer/artikel/Boesartige-Race-Conditions-und-Data-Races-3721793.html. Zuletzt besucht am: 6. Juni 2019.

[22] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf. Zuletzt besucht am: 6. Juni 2019.

[23] GSI. Redmine GSI CbmRoot. https://lxcbmredmine01.gsi.de/. Zuletzt besucht am: 6. Juni 2019.

Bitte dieses Formular zusammen mit der Abschlussarbeit abgeben!

Erklärung zur Abschlussarbeit

gemäß § 25, Abs. 11 der Ordnung für den Bachelorstudiengang Informatik vom 06. Dezember 2010:

Hiermit erkläre ich Herr / Frau

_____

Die vorliegende Arbeit habe ich selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst.

Frankfurt am Main, den

_____

Unterschrift der Studentin / des Studenten