Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Parallelisierung eines Algorithmus zur topologischen Rekonstruktion in Flüssigszintillator-Detektoren

Parallelization of an algorithm for the topological reconstruction in liquid scintillator detectors

— Bachelorarbeit —

Vorgelegt von:	Hauke Ortwin Schmidt
Matrikelnummer:	6541961
Fakultät:	Mathematik, Informatik und Naturwissenschaften
Fachbereich:	Informatik
Studiengang:	Computing in Science, Schwerpunkt Physik

Erstgutachter:	Dr. Björn Wonsak
Zweitgutachterin:	Prof. Dr. Caren Hagner

Hamburg, 10.11.2016

Zusammenfassung

Mit dem JUNO Experiment soll insbesondere das Energiespektrum der Oszillation der Neutrinoflavoureigenzustände sehr genau vermessen werden. Dazu wird ein unterirdischer Detektor gebaut, der im Zentrum aus einer Kugel mit 20 Kilotonnen Flüssigszintillator besteht. Durch die Messungen soll herausgefunden werden, in welcher Massenhierarchie die drei Neutrino-Masseneigenzustände zueinander stehen. Dazu soll ein Algorithmus zur topologischen Rekonstruktion der Ereignisse in Flüssigszintillator-Detektoren eingesetzt werden. Diese Bachelorarbeit thematisiert die Parallelisierung des bestehenden Algorithmus. Dazu wurden verschiedene Ansätze betrachtet, getestet und miteinander verglichen. Schließlich wurde die Parallelisierung mit OpenMP umgesetzt. Die Rechnungen konnten bei Einsatz aller 24 Prozessorkerne der Testumgebung bis zu 12,7 mal schneller ausgeführt werden.

Abstract

With the JUNO experiment the energy spectrum of the oszillation of the neutrinoflavour eigenstates has to be measured very precisely. Therefor an underground detector is build, which consists of a sphere with 20 kilo tonns liquid scintillator in the center. This is done to find out the mass hierarchy of the three neutrino mass eigenstates. Therefor an algorithm for the topological reconstruction of the events in the liquid scintillator detector is used. This bachelor thesis is about the parallelization of the existing algorithm. For this purpose, different approaches have been considered, tested and compared. Finally, the parallelization was implemented with OpenMP. The calculations were carried out up to 12.7 times faster by using all 24 cores of the test environment.

Inhaltsverzeichnis

1. Einleitung

Neutrinos haben Eigenschaften, die jenseits des Standardmodells sind und somit ist ihre Erforschung auch Erforschung neuer Physik. Um die Eigenschaften der Neutrinos genauer untersuchen zu können, ist ein Hochleistungsdedektor erforderlich. Ein solcher Detektor wird aktuell für das Jiangmen Underground Neutrino Observatory (JUNO) gebaut. Mit dem JUNO Experiment soll unter anderem das Energiespektrum der Oszillation der Neutrinoflavoureigenzustände gemessen werden, indem Neutrinos, die in Kernreaktoren entstehen, mit einem Flüssigszintillatordetektor beobachtet werden. Dabei wird bei Reaktionen im Flüssigszintillator Licht ausgesendet, dass in Photomultiplier Tubes (PMT), die rund um den kugelförmigen Detektor angeordnet sind, Signale auslöst. Da die Neutrinos allerdings nur sehr selten mit Materie interagieren, ergeben sich bei sehr großen Detektoren Schwierigkeiten durch Untergrundbelastung durch andere Teilchen, die ebenfalls im Detektor Signale auslösen. Daher müssen die Messdaten des Detektors gefiltert werden, um nur die durch Neutrinos hervorgerufene Positron e^+ und γ-Quanten zu messen. Dies lässt sich über eine Strategie zum Ausschluss anderer Signale (Vetostrategie), die die Rekonstruktion der Spuren im Detektor beinhaltet [7], realisieren. Ein möglicher Algorithmus für eine sehr genaue Rekonstruktion ist die topologische Rekonstruktion. Dieser Algorithmus ist aber extrem rechenintensiv, sodass dieser optimiert werden muss. Ziel ist es, dass der Algorithmus die Ergebnis in Echtzeit produziert. Ein Baustein dieser Optimierung hin zu einer Hochleistungssoftware ist die Parallelisierung. Heutzutage hat jedes Smartphone eine Multiprozessorarchitektur, sodass Programme, die nur auf einem Kern laufen, nicht mehr Stand der Technik sind. Der Normalfall sollte ein parallel ausführbares Programm sein, um die zur Verfügung stehende Hardware auszunutzen. Dies gilt besonders auch für naturwissenschaftliche Software. Ziel dieser Arbeit ist es für das bestehende Programm die Parallelisierungsmöglichkeiten zu untersuchen und eine erste Parallelisierung durchzuführen. Dabei müssen einige Rahmenbedingungen beachtet werden, wie etwa die Konfliktfreiheit paralleler Programmabschnitte. Das Ergebnis darf

zudem nicht durch die Parallelisierung beeinflusst werden. Zur Einordnung in den Kontext wird in Kapitel 2 ein Überblick über die Physik rund um das Neutrino gegeben und das JUNO-Experiment, für das die Software vorgesehen ist, vorgestellt. Die Beschreibung des Aufbaues des Rekonstruktions-Programmes und dessen Algorithmus folgt in Kapitel 3. Anschließend werden in Kapitel 4 die für die Ausführung der Rekonstruktion benötigten Softwarepakete vorgestellt. In Kapitel 5 wird erläutert, wie die Parallelisierung durchgeführt wurde. Die Auswertungen zu dieser Parallelisierung finden sich in Kapitel 6. Kapitel 7 fasst die Ergebnisse noch einmal abschließend zusammen und gibt einen Ausblick auf die weiteren Optimierungsmöglichkeiten des Programmes.

2. Das JUNO-Experiment

In diesem Kapitel wird ein Überblick über den aktuellen Stand der Neutrinophysik gegeben und das Jiangmen Underground Neutrino Observatory (JUNO), das diese weiter erforschen soll, vorgestellt.

2.1. Neutrinophysik

Beim radioaktiven β-Zerfall werden Elektronen e^- oder Positronen e^+ ausgesendet. Allerdings kann es sich nicht wie beim α-Zerfall um einen Zweikörper-Zerfall handeln, denn sonst würde das beobachtete kontinuierliche Energiespektrum eine Verletzung von Energie- und Impulssatz, sowie Drehimpulserhaltung bedeuten [6].

Deshalb postulierte Wolfgang Pauli 1930 in einem offenen Brief, dass beim β-Zerfall ein bisher im Experiment unentdecktes, neutrales Teilchen mit Spin $\hbar/2$ ausgesandt wird, das er vorläufig „Neutron“ nannte [15].

Die Entdeckung des Neutrons, mit einer Masse, die in etwa der Masse des Protons entspricht, durch James Chadwick im Jahr 1932 führte zu der Erkenntnis, dass beim β-Zerfall ein anderes neutrales Teilchen ausgesendet wird [6]. Denn um die maximale Energie im β-Energiespektrum erreichen zu können, muss die Masse des neutralen Teilchens sehr klein sein [18]. Daher wurde das hypothetische Teilchen umbenannt und der heute gebräuchliche Name *Neutrino*[1] eingeführt.

Nach dem Neutrinomodell wird beim β^--Zerfall eines Atom X mit der Massenzahl A und der Ladungszahl Z in ein Atom Y neben dem Elektron auch ein Anti-Neutrino $\bar{\nu}$ ausgesendet [6]:

$$ {}^{A}_{Z}X \rightarrow {}^{A}_{Z+1}Y + e + \bar{\nu} \tag{2.1} $$

[1] *ital. für* kleines Neutron

und beim β^+-Zerfall ein Positron und ein Neutrino:

$$ {}^{A}_{Z}X \rightarrow {}^{A}_{Z-1}Y + e^+ + \nu. \tag{2.2} $$

Allerdings blieb die Suche nach Neutrinos viele Jahre erfolglos. Erst 1955 gelang Frederick Reines und Clyde L. Cowan der experimentale Nachweis [6]. Da die Neutrinos zu den elektrisch neutralen Leptonen gehören, unterliegen sie weder der starken noch der elektromagnischen Wechselwirkung und können in Reaktionen nur indirekt über geladene Teilchen oder Gammaquanten nachgewiesen werden [16]. Es gibt drei verschiedene Neutrinos, jeweils eins in jeder Flavourfamilie. Dies sind das Elektron-Neutrino ν_e, das Myon-Neutrino ν_μ und das Tau-Neutrino ν_τ. Zu jedem dieser Teilchen gibt es ein entsprechendenes Antiteilchen $\overline{\nu}$.

2.1.1. Neutrino-Oszillation

Über die Neutrino-Oszillation kann man indirekt zeigen, dass Neutrinos eine von Null verschiedene Ruhemasse haben. Dazu betrachtet man den periodischen Übergang der Wahrscheinlichkeiten der einzelnen Eigenzustände der Flavour-Familien $|\nu_e\rangle$, $|\nu_\mu\rangle$ und $|\nu_\tau\rangle$. In Experimenten mit solaren Neutrinos misst man nur etwa halb so viele solare Neutrinos wie in Sonnenmodellen ohne Neutrinooszillation vorhergesagt [16]. Dies kann nur geschehen, wenn die Neutrinos eine Masse haben und die Masseneigenzustände Mischungen der Flavoureigenzustände sind. Die Flavour-Oszillation kann deshalb als sehr mächtige Eigenschaft angesehen werden, die dabei hilft die wesentlichen Eigenschaften der Neutrinos und anderer neuer Physik zu untersuchen [2]. Die Oszillation kann man in den Neutrinostrahlen von Reaktoren und Beschleunigern nachweisen oder man kann sie durch die Messung von Neutrinos, die in der Sonne oder in der Erdathmosphäre entstehen, beobachten. Es gibt drei Masseeigenzustände $|\nu_1\rangle$, $|\nu_2\rangle$ und $|\nu_3\rangle$, die sich durch die Mischung der Neutrinoflavoureigenzustände ergeben und nicht einem einzelnen Flavour-Eigenzustand zugeordnet werden können.
Unter der Annahme, dass es drei Flavoureigenzustände gibt, kann die Mischung durch die folgende Matrixschreibweise dargestellt werden:

$$\begin{pmatrix} \nu_e \\ \nu_\mu \\ \nu_\tau \end{pmatrix} = \begin{pmatrix} U_{e1} & U_{e2} & U_{e3} \\ U_{\mu 1} & U_{\mu 2} & U_{\mu 3} \\ U_{\tau 1} & U_{\tau 2} & U_{\tau 3} \end{pmatrix} \begin{pmatrix} \nu_1 \\ \nu_2 \\ \nu_3 \end{pmatrix} \tag{2.3}$$

dabei ist ν_i ein Masseeigenzustand für die Masse m_i eines Neutrinos (für $i = 1, 2, 3$). U ist die Maki-Nakagawa-Sakata-Pontecorvo (MNSP) Matrix, die Mischungsmatrix der Neutrinos.

Nimmt man an, dass die MNSP Matrix U unitär ist, dann kann man die Matrix mit drei Flavourmischungswinkel und drei Phasen für die CP-Verletzung parametrisieren:

$$U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_{23} & s_{23} \\ 0 & -s_{23} & c_{23} \end{pmatrix} \begin{pmatrix} c_{13} & 0 & s_{13}e^{-i\delta} \\ 0 & 1 & 0 \\ -s_{13}e^{i\delta} & 0 & c_{13} \end{pmatrix} \begin{pmatrix} c_{12} & s_{12} & 0 \\ -s_{12} & c_{12} & 0 \\ 0 & 0 & 1 \end{pmatrix} P_\nu \tag{2.4}$$

mit $c_{ij} \equiv \cos\theta_{ij}$ und $s_{ij} \equiv \sin\theta_{ij}$ für $ij = 12, 13, 23$ und $P_\nu = \mathrm{Diag}\{e^{i\rho}, e^{i\sigma}, 1\}$.
Mit den Werten der Mischungsmatrix können dann die Übergangswahrscheinlichkeiten nach folgender Formel berechnet werden[2]:

$$P(\nu_\alpha \to \nu_\beta) = \frac{\sum_i |U_{\alpha i}|^2 |U_{\beta i}|^2 + 2\sum_{i<j} \left[\mathrm{Re}(U_{\alpha i} U_{\beta j} U^*_{\alpha j} U^*_{\beta i}) \cos\Delta_{ij} - \mathrm{Im}(U_{\alpha i} U_{\beta j} U^*_{\alpha j} U^*_{\beta i}) \sin\Delta_{ij}\right]}{(UU^\dagger)_{\alpha\alpha}(UU^\dagger)_{\beta\beta}} \tag{2.5}$$

wobei $\Delta_{ij} \equiv \frac{(m_i^2 - m_j^2)L}{2E}$ ist, mit der zurückgelegten Strecke L und der Energie E des Teilchens.
Im Standard drei-Flavour Modell kann man die Neutrinooszillation mit sechs voneinander unabhängigen Parametern beschreiben[2]. Dies sind die quadrierten Massendifferenzen Δm^2_{21} und Δm^2_{31}, die Flavourmischungswinkel θ_{12}, θ_{13} und θ_{23} und die Phase δ der Dirac CP-Verletzung. Aus den bisherigen Experimenten konnte man die Flavourmischungswinkel, die quadrierte Massendifferenz Δm^2_{21} und den Betrag der quadrierten Massendifferenz Δm^2_{31} sehr genau bestimmen. Bisher fehlen noch Werte für das Vorzeichen der quadrierten Massendifferenz Δm^2_{31} und ein Wert für die CP-Verletzung δ.
In Tabelle 2.1 sind die aktuell (Stand Oktober 2016) besten Mittelwerte für die Parameter aufgelistet, dabei ist Δm^2 definiert als $\Delta m^2 = m_3^2 - (m_2^2 + m_1^2)/2$.
Unter anderem sollen diese Werte mit JUNO weiter verfeinert werden.

Parameter	bester Mittelwert	3σ
$\Delta m^2_{21}[10^{-5}eV^2]$	7.37	6.93 - 7.97
$\lvert\Delta m^2[10^{-3}ev^2]$	2.50 (2.46)	2.37 - 2.63 (2.33 - 2.60)
$\sin^2\theta_{12}$	0.297	0.250 - 0.354
$\sin^2\theta_{23}, \Delta m^2 > 0$	0.437	0.379 - 0.616
$\sin^2\theta_{23}, \Delta m^2 < 0$	0.569	0.383 - 0.637
$\sin^2\theta_{13}, \Delta m^2 > 0$	0.0214	0.0185 - 0.0246
$\sin^2\theta_{13}, \Delta m^2 < 0$	0.0218	0.0186 - 0.0248
δ/π	1.35 (1.32)	(0.92-1.99)((0.83-1.99))

Tabelle 2.1.: Beste Mittelwerte für die Parameter der Oszillation mit dem 3σ Intervall, basierend auf einer globalen Analyse [8]

Eine der ungelösten Fragen der Neutrinophysik ist, ob die normale Massenhierarchie ($m_1 < m_2 < m_3, \Delta m^2 > 0$) oder die invertierte Massenhierarchie ($m_3 < m_1 < m_2, \Delta m^2 < 0$) (siehe Abbildung 2.1) der Neutrinos gilt. Um das herauszufinden, muss gemessen werden, welches Vorzeichen die quadrierte Massendifferenz $\Delta m^2_{31} = m^2_3 - m^2_1$ hat.
Das JUNO Experiments möchte dies herauszufinden, indem die Teilchenflussdichte und das Energiespektrum der $\overline{\nu}_e \to \overline{\nu}_e$ Oszillation von Reaktorantineutrinos sehr genau gemessen wird. Andere Experimente, wie INO oder PINGU, beobachten dazu die Oszillation von atmosphärischen (Anti-)Neutrions $\nu_\mu \to \nu_\mu$ [4][5].

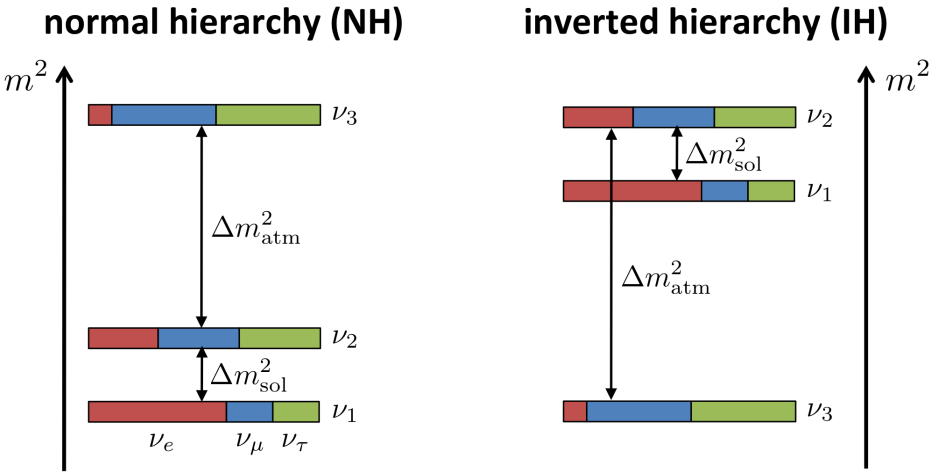

Abbildung 2.1.: Die beiden möglichen Hierarchien der Neutrino Masseneigenzustände, links die normale Massenhierarchie mit einem schweren ν_3, rechts die invertierte Massenhierarchie mit einem leichten ν_3 [11].

2.2. JUNO Detektor

Das JUNO Experiment ist ein Neutrino Experiment in der Nähe der Stadt Jiangmen im Süden Chinas. Die Massenhierarchie soll mit Hilfe von zwei Neutrinostrahlen, die von zwei jeweils 53 Kilometer entfernten Kernkraftwerken ausgesandt werden, untersucht werden. Dazu wird ein Detektor 700 Meter unter der Erde gebaut, der in der Mitte aus einer Kugel mit 20 Kilotonnen Flüssigszintillator besteht. Geplant ist mit den Messungen 2020 zu beginnen.

Wie in Abbildung 2.2 zu sehen ist, besteht der JUNO Detektor aus dem zentralen Flüssigszintillatordetektor einem Wasser-Cherenkov Detektor und einem Myonentracker. Das zylindrische Wasserbecken um den zentralen Detektor sorgt dafür, dass die natürliche Radioaktivität des umgebenen Gesteins und der Luft abgeschirmt wird. Gleichzeitig fungiert das Wasserbecken als Veto Detektor, in dem mit etwa 2000 Photomultiplier Tubes[2] (PMTs) das Cherenkovlicht kosmischer Myonen detektiert wird, die einen Untergrund für den eigentlichen Detektor darstellen. Cherenkovlicht entsteht, wenn sich gelandene Teilchen in einem Material schneller bewegen, als Licht in diesem Material. Die kosmischen Myon reagieren im Detektor in $\beta - n$ Zerfällen, bei denen ^{8}He und ^{9}Li entstehen, die ähnliche Signale wie inverse β-Zerfälle von Reaktorantineutrinos auslösen[2].

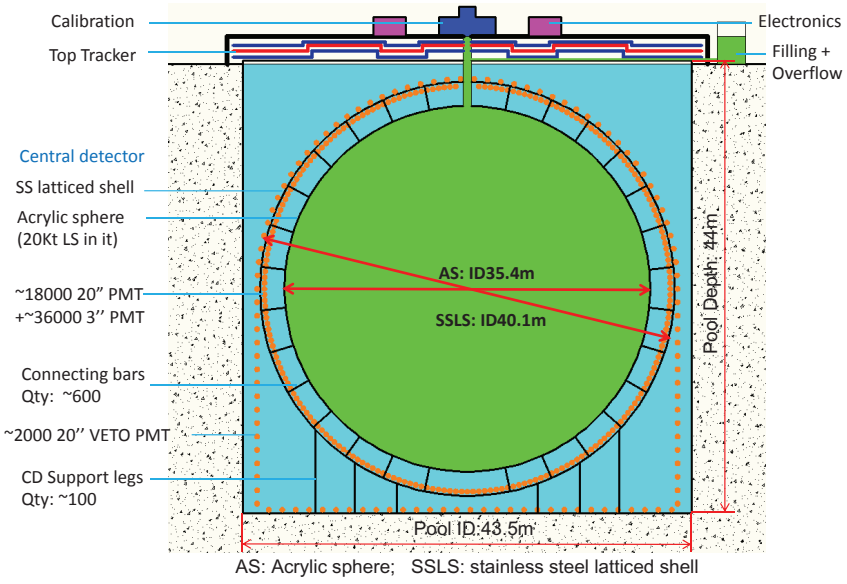

Abbildung 2.2.: Schematischer Aufbau des JUNO Detektors mit dem Flüssigszintillatordetektor (grün) in der Mitte, den PMTs rundherum und dem Myonentracker oberhalb davon [12](Zahlen aktualisiert).

[2] *engl. für* Photoelektronenvervielfacher Röhren

Oberhalb des Wasserbeckens ist ein weiterer Myonentracker, der aus dem OPERA Experiment stammt und den genauen Weg der Myonen erfassen soll[2]. Rund um den kugelförmigen Flüssigszintillatordetektor werden etwa 18.000 PMTs mit einem Durchmesser von 20 Inch und bis zu 36.000 PMTs mit einem Durchmesser von 3 Inch installiert (Stand Oktober 2016).

Im JUNO Experiment möchte man inverse β-Zerfälle eines Anti-Elektron-Neutrinos $\bar{\nu}_e$ und eines Protons p in ein Positron e^+ und ein Neutron n beobachten[2]:

$$\bar{\nu}_e + p \rightarrow e^+ + n. \tag{2.6}$$

Das Positron gibt seine Energie sehr schnell mit zwei γ-Quanten mit je 511 keV ab und löst damit unmittelbar ein Signal im Detektor aus. Das Neutron wird nach etwa 200 μs von einem Positron eingefangen und gibt dann einen γ-Quanten mit 2.2 MeV[3] ab. Diese γ-Quanten regen die Moleküle des Flüssigszintillators an, die die Anregungsenergie dann als Licht wieder abgeben, sodass es von den PMTs registiert werden kann[17].

Die hohe Anzahl an PMTs soll eine fast vollständige Abdeckung der Kugeloberfläche und damit zusammen mit einem besonders leistungsfähigen Flüssigszintillator eine besonders große Energieauflösung von 3% bei 1 MeV Neutrinoenergie [7] garantieren. Diese Energieauflösung wird benötigt, um eine eindeutige Zuordung der Massenordung bestimmen zu können. In Abbildung 2.3 ist für die beiden möglichen Massenordnungen der erwartete Fluss an Antineutrions aus Reaktoren aufgetragen in der Abhängigkeit vom Quotienten aus Wegstreche der Teilchen L und ihrer Energie E. Die Entfernung zu den beiden Kernkraftwerken wurde genau so gewählt, dass mit $\frac{L}{E} \sim 11$ für die Maxima der beiden möglichen Oszillationen der Neutrinos eine maximale Verschiebung zueinander erwartet wird[2]. Die oszillationsbedingte Überlebenswahrscheinlichkeit für die Elektron-Anti-Neutrinos beim JUNO ist nach [7] näherungsweise gegeben durch

$$P(\bar{\nu}_e \rightarrow \bar{\nu}_e) = 1 - \cos^4\theta_{13}\sin^2 2\theta_{12}\sin^2\Delta_{21} - \sin^2 2\theta_{13}\left[\cos^2\theta_{12}\sin^2\Delta_{31} + \sin^2\theta_{12}\sin^2\Delta_{32}\right] \tag{2.7}$$

mit

$$\Delta_{ij} = \frac{\Delta m^2_{ij} L}{4E}. \tag{2.8}$$

[3]Mega-Elektronen-Volt: $1,602176 \cdot 10^{-13}$ J

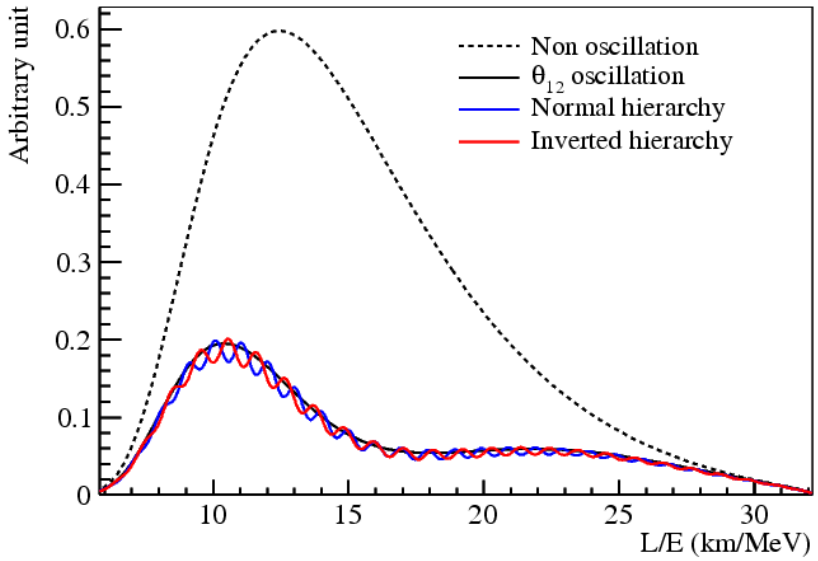

Abbildung 2.3.: Muster der $\bar{\nu}_e \rightarrow \bar{\nu}_e$-Oszillations in Abhängigkeit von der Massenordnung, aufgetragen über dem Quotienten aus Entfernung und Teilchenenergie. Das JUNO Experiment befindet sich bei $\frac{L}{E} \sim 11$ [2]

2.3. 3D Rekonstruktion im JUNO Experiment

Um aus den Messdaten des Experiments die einzelnen Ereignisse zu rekonstruieren, gibt es verschiedene Ansätze, die verschiedene Ziele verfolgen. Eines der Ziele ist es, aus den Ereignissen die durch Untergrund hervorgerufenen Ereignisse herauszufiltern. Da der JUNO Detektor sehr groß und wesentlich dichter an der Erdoberfläche als vorherige Neutrino-Experimente sein wird, rechnet man mit einem sehr großen Untergrund an durch Myonen hervorgerufenen Ereignissen im Detektor. Dieser wird ohne Vetostrategie mit 71 Ereignissen pro Tag in etwa so groß sein wie das Signal durch Reaktorantineutrinos mit 73 inversen Betazerfällen pro Tag [2]. Daher wird eine Veto-Strategie benötigt, die nicht den kompletten Detektor blind für den Nachweis von Reaktorantineutrinos macht, sondern nur einen kleinen Korridor rund um die Spur des Myons. Deshalb müssen die Spuren der Myonen im Detektor rekonstruiert und analysiert werden. Um die Spur zu rekonstruieren, gibt es mehrere Methoden, die auf unterschiedlichen Informationen aus dem Detektor aufbauen, etwa auf den Zeiten der ersten Treffer an dem PMTs oder auf den Maxima der Ladungen [13]. Die Forschungsgruppe Neutrinophysik in Hamburg [7] versucht hierfür ebenfalls eine Methode zu implementieren. Derzeit wird dafür die Wonsak Rekonstruktion (siehe Kapitel 3) unter anderem für diesen Anwendungsfall weiterentwickelt. Besondere Herausforderungen für die bisherigen Algorithmen stellen dabei Myonenbündel

und schauernde Myonen da, die mit der topologischen Rekonstruktion berechnet werden können. Andererseits soll der topologische Algorithmus aber auch dazu verwendet werden, Ereignisse mit niedrigeren Energien sicher zu rekonstruieren. Dazu muss der Algorithmus robust gegenüber einer Fluktuation in den Signalen sein. Die bisherigen Algorithmen erzielen meist ihre Ergebnisse in sehr kurzer Zeit. Die topologische Rekonstruktion hingegen benötigt hierfür aktuell noch sehr lange. Daher soll die Rechenzeichenzeit bei gleichbleibend hoher Robustheit und Präzision der Ergebnisse verkürzt werden.

3. Wonsak Rekonstruktion

In diesem Kapitel wird das für diese Arbeit zugrundegelegte, sequentielle Programm, dass parallelisiert wurde, kurz beschrieben.
Bei der Wonsak Rekonstruktion handelt es sich um einen Algorithmus, der durch topologische Methoden versucht, Teilchen und ihre Spuren in einem Detektor, der mit Flüssigszintillator gefüllt ist, zu rekonstruieren. Die Rekonstruktion ist in C++ geschrieben und greift auf einige C++-Bibliotheken zurück, die im nachfolgenden Kapitel beschrieben sind.
Die Rekonstruktion baut auf der Idee auf, dass man den Ort einer einzelnen punktförmigen Lichtquelle mit Hilfe der Zeitunterschiede der Lichterfassung in den Photomultiplier Tubes des Detektors errechnen kann. Für eine Lichtspur funktioniert dies aber nicht so einfach, da es keine Korrelation zwischen Signal und Aussendezeit gibt. Wenn man aber annimmt, dass sich die Teilchen auf einer geraden Spur mit Lichtgeschwindigkeit bewegen und man einen Referenzpunkt auf der Spur mit der Referenzzeit kennt, kann man daraus die Zeit rekonstruieren, die ein Signal bis zum PMT benötigt.

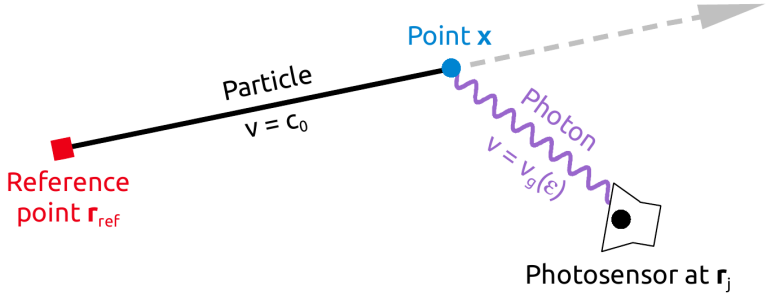

Abbildung 3.1.: Spurrekonstruktion mit Referenzpunkt r_{ref} und Emissionspunkt x [13]

In Abbildung 3.1 ist eine Möglichkeit schematisch aufgetragen, wie Referenzpunkt r_{ref}, Emissionspunkt x, Photomultiplier Tube am Ort r_j und Teilchen-Spur zueinander liegen. Die Zeit bis zur Lichterfassung des Signals kann dann mit folgender Formel berechnet

werden [13]:

$$t(x) = t_{ref} \pm \underbrace{\frac{|x - r_{ref}|}{c_0}}_{particle} + \underbrace{\frac{|r_j - x|}{v_g(\epsilon)}}_{photon} . \tag{3.1}$$

Dabei ist $t(x)$ die Zeit bis zur Detektion des Lichtes, t_{ref} die Referenzzeit, c_0 die Lichtgeschwindigkeit im Vakuum und $v_g(\epsilon)$ die Lichtgeschwindigkeit im Medium. Diese Formel hat keine eindeutige Lösung, sondern ergibt, wie in Abbildung 3.2 zu sehen ist, Ellipsoide mit der Wahrscheinlichkeitsverteilung für den Emissionspunkt rund um den PMT.

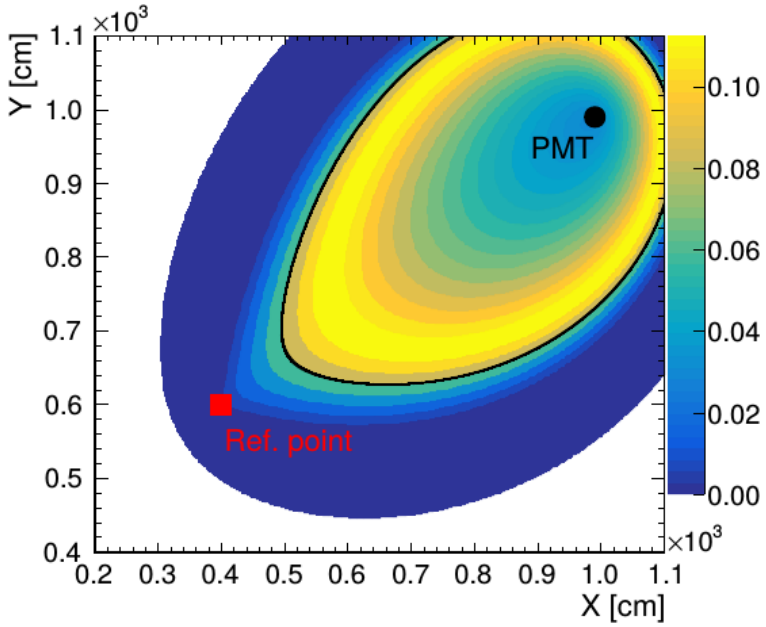

Abbildung 3.2.: Zweidimensionle Projektion der Wahrscheinlichkeitsverteilung für den Ort der Lichtquelle in Abhängigkeit von Referenzpunkt und PMT Position [13]

Diese Wahrscheinlichkeitsverteilung wird für jeden PMT des Detektors berechnet und die Ergebnisse kombiniert, indem das Detektorvolumen durch ein Gitternetz dargestellt wird, an dessen Gitterpunkten die Wahrscheinlichkeiten aufsummiert werden. Dadurch ergibt sich aus der Überlagerung der Ellipsoide die Spur mit der höchsten Wahrscheinlichkeit.
Die Informationen, die man aus einem Durchlauf dieser Rechnungen erhält, reichen aber oft noch nicht aus, um die Spur exakt zu rekonstruieren. Daher werden die Rechnungen wiederholt, dann aber wird das vorherige Ergebnis benutzt, um die dann berechneten Wahrscheinlichkeiten zu gewichten.
Mit einer Strategiedatei wird vom Benutzer bestimmt, wieviele dieser Wiederholungen durchgeführt werden und wie groß dabei die Abstände zwischen den Gitternetzpunkten sein sollen.

3.1. Gliederung der Wonsak Rekonstruktion

Das Softwareprojekt gliedert sich in einen Bibliotheken-Teil und einen Programmteil, der die eigentliche Rekonstruktion enthält. Der Bibliotheken-Teil gliedert sich wiederum in vier Bibliotheken:

Detector-Event definiert Objekte von Detektoren, Materialien, Ereignissen, PMTs und Treffern von PMTs

Mesh definiert Objekte des Gitternetzes innerhalb des Detektors inklusive Blöcken und Zell-Iterator

PMT-Signal definiert die gewichtete Signalfunktion, die zur Berechnung der Rekonstruktion notwendig ist

Utilities definiert weitere Hilfsmethoden, wie etwa das Auslesen der Argumente für die Rekonstruktion aus der Konfigurationsdatei, einige angepasste Fehlermeldungen und die Lookup-Tables[1] für die Berechnung der Lichtverteilung im Detektor.

Der Programmteil beinhaltet den eigentlichen Algorithmus, die Methoden, die die Strategiedatei einlesen und in die Strategie zur Rekonstruktion umsetzen, Methoden zum Finden von Referenzzeit und -punkt und die Steuerung der Rekonstruktion.

3.2. Algorithmus der Wonsak Rekonstruktion

Die Rekonstruktion kann mit zwei unterschiedlichen Algorithmen ausgeführt werden. Zum einen ist dies die Rekonstruktion des detektierten Lichtes (`DetectedLightRecoAlgo`), zum anderen die Rekonstruktion des emittierten Lichtes (`EmittedLightRecoAlgo`). Je nach Konfiguration wird entweder der eine oder der andere Algorithmus ausgeführt. Dabei kann der Algorithmus auf unterschiedliche Gitternetzweiten ausgeführt werden. In der Regel wird die Konfiguration so gewählt, dass der Algorithmus mehrfach hintereinander aufgerufen wird. Dabei wird zuerst ein gröberes Netz verwendet, das dann in der darauffolgenden Iteration in den Bereichen mit detektierten Licht verfeinert wird. Zudem wird das Ergebnis

[1] *engl. für* Nachschlagetabellen: komplexe Rechnungen werden durch das Nachschlagen schon berechneter Werte ersetzt

der vorherigen Iteration als Wahrscheinlichkeitsmaske für die folgende Iteration verwendet, um das Ergebnis zu gewichten.

DetectedLightRecoAlgo Die Rekonstruktion des detektierten Lichtes besteht im Wesentlichen aus der Addition der detektierten Photonen im Gitternetz. Dazu werden in der Methode `DoRun` die Photomultiplier Tubes der Reihe nach auf Treffer überprüft und bei einem Treffer wird die Signalfunktion berechnet und die berechneten Daten in das Gitter gefüllt.

EmittedLightRecoAlgo Die Rekonstruktion des emittierten Lichtes setzt sich aus der Rekonstruktion des detektierten Lichtes, der Berechnung des emittierten Lichtes und der Überprüfung der Ergebnisse mit den Informationen der nicht getroffenen PMTs zusammen. Der erste Teil läuft sehr ähnlich wie der `DetectedLightRecoAlgo` ab, wobei auch die nicht getroffenen PMTs betrachtet werden. Da jeder PMT über seine ID referenzierbar ist, wird diese für einen nicht getroffenen PMT in einem Vektor gespeichert, sodass im weiteren Verlauf des Algorithmus darauf zugegriffen werden kann. Zudem wird der Beitrag des PMT zur räumlichen Lichtdetektionseffizienz in das Gitternetz eingefügt. Im Anschluss wird zellenweise für jeden Block des Gitters die Anzahl der detektierten Szintillationsphotonen durch die Lichterfassungseffizienz der Zelle geteilt, um eine Approximation für die räumliche Verteilung der Dichte der emittierten Szintillationsphotonen zu bekommen. Im dritten Schritt, der allerdings nur in der vorletzten Iteration ausgeführt wird, wird die Methode `EvaluateResults` aufgerufen, die mit den Informationen aus den nicht getroffenen PMTs das Ergebnis evaluiert. Für jeden nicht getroffenen PMT wird für jede Zelle die Poisson Wahrscheinlichkeit berechnet, dass der PMT keinen Treffer aus dieser Zelle erhalten hat, basierend auf einer Anzahl an erwarteten Treffern. Schließlich wird diese Wahrscheinlichkeit mit dem Wert in der Gitterzelle multipliziert.

4. Weitere Softwarekomponenten

In diesem Kapitel werden die weiteren Software-Komponenten, die bei dieser Arbeit benutzt wurden und für die Ausführung der Rekonstruktion notwendig sind, beschrieben. Außerdem werden die Grundlagen der parallelen Programmierung erläutert.

4.1. ROOT

Das ROOT Framework[1], entwickelt am CERN, ist ein multifunktionales Werkzeug, das sich in C++ einbinden lässt. Es verfügt über eine Vielzahl an Funktionalitäten, die in der Teilchenphysik eingesetzt werden können. Zudem definiert es ein eigenes Datenformat, dass sowohl zur Eingabe, als auch für die Ausgabe der Daten in der Wonsak-Rekonstruktion verwendet wird. Die Plots in dieser Arbeit wurden mit Hilfe der ROOT Klassen `TGraph` und `TMultiGraph` erstellt.

4.2. BOOST

Die BOOST C++ Bibliothek ergänzt den Standardumfang von C++ um einige hilfreiche Klassen, wie z.B. die intelligenten Pointer[2], die eine sichere Handhabung ermöglichen und gegenüber herkömmlichen Pointern den Vorteil bieten, dass sie vor dem Löschen geschützt sind, wenn sie noch verwendet werden. Diese intelligenten Pointer werden auch in der Rekonstruktion verwendet.

4.3. GCC Compiler

Um das Programm von C++ in ausführbaren Maschinencode zu übersetzen, wurde der Compiler der GNU Compiler Collection (gcc) verwendet. Die benutzte Version 5.4.0

[1] *engl. für* Rahmenstruktur
[2] *engl. für* Zeiger: zeigt auf eine Adresse im Speicher

implementiert den C++11-Standard und den OpenMP-Standard 4.0 (siehe 4.4.2).

4.4. Parallele Programmierung

Heutige Rechnersysteme haben eine Mehrkern-Prozessor-Architektur. Um diese nutzen zu können, bedarf es spezieller Programmfunktionen. Diese sorgen dafür, dass ein Prozess in mehrere sogenannte Threads aufgeteilt wird. Bei einem Thread handelt es sich eine um sequentielle Abfolge von zusammengehörigen Befehlen. Dabei besteht jedes Programm aus mindestens einem Hauptthread, der das Programm initialisiert und entweder weitere Threads erzeugt, die Unteraufgaben ausführen, oder selbst die Arbeit erledigt [1]. Die Threads teilen sich die Ressourcen des Systems. Daher muss darauf geachtet werden, dass sich die Threads nicht gegenseitig beeinflussen können. Da dies nicht nur auf den eigenen Programmcode zutreffen muss, sondern auch auf alle benutzten Bibliotheks- und Systemfunktionen, ist beim Programmieren erhöhte Aufmerksamkeit gefordert. Alle parallel auszuführenden Funktionen müssen „thread-safe“[3] sein. Ansonsten muss das thread-safe-Verhalten selbst hergestellt werden [3]. Im Nachfolgenden werden zwei Methoden zur Threaderzeugung beschrieben, die in C++ verwendet werden können.

4.4.1. Posix-Threads

Bei Posix-Threads handelt es sich um eine portable Threading-Bibliothek, die sich als Standard für Linux durchgesetzt hat [1]. Posix-Threads erlauben dem Anwender direkt zu steuern, wie das Programm die Threads einsetzt, um Parallelität zu erzeugen. Da es sich allerdings um ein Element aus C handelt, dass nur aus Kompatibilitätsgründen auch in C++ genutzt werden kann, gibt es einige Schwierigkeiten in der Verwendung in größeren C++ Projekten.

4.4.2. OpenMP

Um dem Programmierer eine einfachere Schnittstelle zur Parallelisierung zu ermöglichen, wurde 1997 von verschiedenen Hardware- und Compilerherstellern der OpenMP-Standard definiert [1]. Bei OpenMP braucht sich der Programmierer nicht um die Erzeugung, Synchronisierung, Lastverteilung und Zerstörung von Threads kümmern, sondern kann dies

[3] *engl. für* sicher parallel ausführbar

dem Compiler überlassen, der mit Hilfe von Steuerbefehlen, sogenannten Pragmas, die Stellen zur Parallelisierung angezeigt bekommt. OpenMP wird hauptsächlich schleifenbasiert zur Parallelisierung eingesetzt, das bedeutet, dass die einzelnen Schleifendurchläufe auf die Threads aufgeteilt werden. Standardmäßig entscheidet OpenMP selbst, wie viele Threads erzeugt werden, um eine optimale Aufteilung zu erreichen. Damit Schleifen auf Threads aufgeteilt werden können, müssen nach OpenMP-Standard 4.0 [14] folgende Bedingungen erfüllt werden:

- die Schleifenvariable muss entweder vom Typ Integer oder eine Variable eines Zufallsiterators sein
- die Schleifenvariable muss mit $<, <=, >$ und $>=$ vergleichbar sein
- die Schleifenvariable muss mit einer Integer-Addition oder -Subtraktion mit einem schleifeninvarianten Wert inkrementiert bzw. dekrementiert werden

Dabei sollten die einzelnen Schleifendurchläufe keine Abhängigkeiten untereinander haben.

5. Durchführung der Parallelisierung

In diesem Kapitel werden die Aufteilung der Daten in der Parallelisierung und verschiedene Ansätze zur Realisierung der Parallelisierung, sowie die schlussendlich umgesetzte Parallelisierung mit Hilfe von OpenMP beschrieben.

5.1. Datenaufteilung

Um die Parallelisierung durchführen zu können, musste zunächst untersucht werden, an welchen Stellen dies sinnvoll ist. Dazu wurde das GNU Performance Analyse Tool `gprof` verwendet, das während der Ausführung des Programmes für jede Methode die Ausführungszeit misst und so die zeitaufwändigsten Abschnitte der sequentiellen Implementation ermittelt.

index	% time	self	children	called	name
1	94.6	0.00	276.21	main	
2	94.6	0.00	276.21	1	WonsakRecoApplication::Run
3	94.6	0.00	276.19	1	RecoManager::ReconstructEvents
4	94.6	0.00	276.19	1	RecoManager::DoEventLoop
5	94.6	0.00	276.18	1	LenaMCRecoStrategy::DoEventReco
6	94.6	0.00	276.18	1	LenaMCRecoStrategy::StartEventReco
7	92.5	0.00	270.30	1	RecoAlgorithm::Run
8	92.5	0.00	270.30	1	EmittedLightRecoAlgo::DoRun
9	81.8	14.15	224.69	29262	EmittedLightRecoAlgo::AddLightDetEffContribution
18	5.2	0.00	15.22	1280	RecoAlgorithm::AdaptCellContentArgs
21	5.1	1.41	13.51	1280	EmittedLightRecoAlgo::FillMeshFromPmtData

Tabelle 5.1.: Ausschnitt: Messergebnis von `gprof`

Die Tabelle 5.1 zeigt einen Ausschnitt aus dem Messergebnis mit `gprof`. Dabei wird in der Spalte *% time* der zeitliche Anteil dieser (*self*) und aller in ihr aufgerufenen Methoden (*children*) an der Gesamtlaufzeit und in der Spalte *called* die Anzahl der Aufrufe dieser Methode dargestellt. Die Zeiten sind jeweils in Sekunden angegeben. Der Tabelle ist zu

entnehmen, dass es sich bis zur Methode `DoRun` (Index 8) um eine Kette an Methodenaufrufen handelt, die jeweils nur einmal ausgeführt werden. Die Methode `DoRun`, die im wesentlichen aus einer Schleife besteht, die für jeden einzelnen PMT die Rekonstruktion aufruft, umschließt somit einen Großteil der rechenzeitintensiven Methodenaufrufe. Da die Berechnungen unabhängig voneinander für jeden PMT ausführbar sind, bot es sich für alle nachfolgenden Ansätze an, an dieser Stelle die Berechnungen der einzelnen PMTs auf die zur Verfügung stehenden Threads aufzuteilen.

5.2. Ansätze zur Parallelisierung

5.2.1. Posix-Threads

Zunächst wurde versucht die Rekonstruktion mit Hilfe von Posix-Threads (siehe Abschnitt 4.4.1) zu parallelisieren. Da es sich bei Posix-Threads allerdings um ein Konstrukt aus C und nicht aus C++ handelt, ergaben sich größere Schwierigkeiten durch die Verwendung von Namespaces [1] in C++. Daher wurde dieser Ansatz nicht weiter verfolgt.

5.2.2. Boost::Threads

Das Boost Framework, das bereits in der Rekonstruktion verwendet wird, bietet ebenfalls eine Threadimplementation an, die auf Posix-Threads aufbaut, aber an C++ angepasste Aufrufe der Threadfunktionen bietet, sodass damit auch eine parallele Implementation gelang. Allerdings führte die statische Zuteilung von gleich vielen Schleifeniterationen an jeden Thread dazu, dass diese unterschiedlich schnell mit ihren Iterationen fertig waren und auf den langsamsten warten mussten. Wie in Abbildung 5.1 zu sehen ist, ergab sich daher schnell eine große Abweichung des Speedup [2], dem Quotient aus sequentieller und paralleler Laufzeit, vom Idealfall. Der ideale Speedup wäre eine linear-antiproportionale Abhängigkeit von Laufzeit und der Anzahl der Threads und ist ein Maß für den Erfolg der Parallelisierung.
Dies zeigt, dass eine Beschleunigung durch Parallelisierung möglich ist, aber eine bessere Datenaufteilung benötigt wird, da die Laufzeiten für die Berechnungen zu einem getroffenen PMT im Vergleich zu einem nicht getroffenen PMT stark abweichen. Eine dynamische

[1] *engl. für Namensraum*: Struktur zur eindeutigen Benennung von Objekten in einem Softwareprojekt
[2] *engl. für* Beschleunigung

Aufteilung wäre möglich, wurde aufgrund der Komplexität einer dynamischen Zuteilung und der alternativen Parallelisierung mit OpenMP nicht umgesetzt.

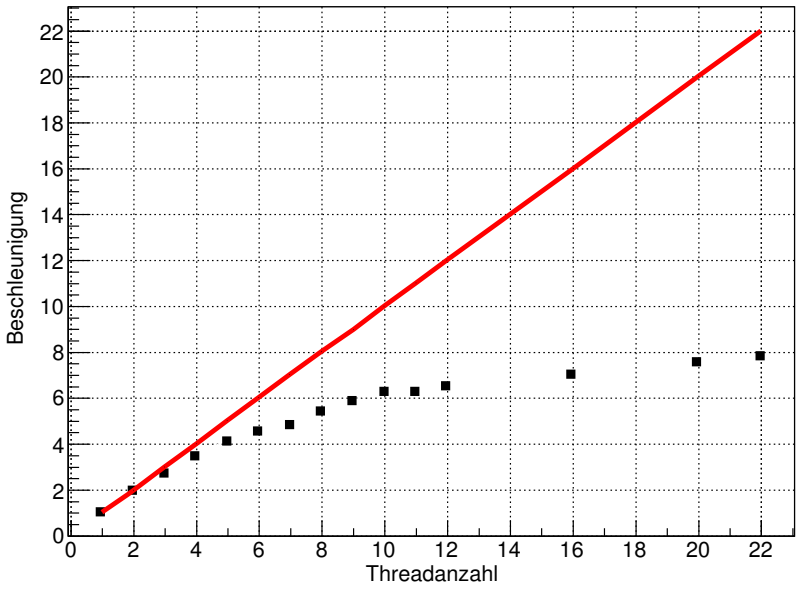

Abbildung 5.1.: Beschleunigung bei einer Iteration der Rekonstruktion in Abhängigkeit von der Anzahl der Threads bei Parallelisierung mit Boost::Threads. Die rote Linie beschreibt die ideale Beschleunigung, die man erreichen möchte

5.2.3. OpenMP

OpenMP (siehe Abschnitt 4.4.2) bietet als compilerbasierter Ansatz zahlreiche Vorteile, wie etwa eine dynamische Aufteilung der Daten zur Laufzeit und eine Parallelisierung ohne Umstrukturierung des Programmes und wurde daher, wie im folgenden Abschnitt beschrieben, in dieser Arbeit eingesetzt.

5.3. Umsetzung mit OpenMP

Aufgrund der beschriebenen Probleme mit Posix-Threads und Boost::Threads wurde die Parallelisierungen mit OpenMP umgesetzt. Nachfolgend wird zunächst beschrieben, welche Vorbereitungen dafür durchgeführt werden mussten und dann die Umsetzung der Parallelisierung für die einzelnen Algorithmen und die dafür benötigten Umstrukturierung zum Erreichen einer thread-sicheren Implementation dargestellt. Die in dieser Arbeit durchgeführten Änderungen sind in den Programm-Code Ausschnitten jeweils `fett` geschrieben.

5.3.1. Vorbereitung

Um auf dem Workgroup-Server, der als Testumgebung diente, OpenMP benutzen zu können, musste zunächst eine aktuellere Version des gcc Compilers installiert werden, da OpenMP erst ab gcc 4.2 korrekt unterstützt wird. Zudem musste das Programm an den C++11-Standard angepasst werden, sodass an einigen Stellen kleinere Modifikationen durchgeführt wurden, die im Weiteren nicht gesondert aufgeführt werden, da es sich lediglich um zusätzliche Klammern um negierte Wahrheitsausdrücke herum, sowie und um eine explizite Typenumwandlung handelt statt einer impliziten Typenumwandlung.

5.3.2. EmittedLightRecoAlgo

Der Algorithmus zur Rekonstruktion des emittieren Lichtes (siehe Abschnitt 3.2) definiert zwei Methoden mit jeweils einer Schleife über alle PMTs und somit die zeitaufwändigsten Abschnitte des Programmes, die hier parallelisiert wurden.
Wie im Abschnitt 5.1 beschrieben, wurden die Berechnungen für die einzelnen PMTs auf die Threads aufgeteilt. Dazu wurde die For-Schleife in der Methode `DoRun` im `EmittedLightRecoAlgo`, die über alle PMTs läuft und die einzelnen Berechnungen startet, mit OpenMP parallelisiert. Hierzu wird zunächst, wie im Listing 5.3 in Zeile 3 zu sehen ist, die Anzahl der Threads mittels einer Variablen festgelegt, die in den Laufargumenten des Rekonstruktionsalgorithmus hinterlegt ist. Dafür wurde dem Kommandozeilenparser ein weiterer Parameter `threads` hinzugefügt (siehe Listing 5.1), um beim Starten des Programmes in der Kommandozeile die Anzahl übergeben zu können.

```
1         // >>> threads (optional)
2         if(vm_.count("threads"))
3             args.SetNumberOfThreads(vm_["threads"].as<size_t>());
```

Listing 5.1: CMDLParser: Optionaler Parameter für die Anzahl der Threads

Ein Programmaufruf in der Kommandozeile mit diesem Parameter sieht dann wie folgt aus:

```
1 ./bin/WonsakReconstruction -c config.cfg -i data.root -o out.root -t 6
```

Wird der Parameter nicht angegeben, so wird die Rekonstruktion mit nur einem Thread ausgeführt (siehe Listing 5.2).

```
1        // change to default threadnumber if required
2        if(numberofthreads_ <= 0)
3            numberofthreads_ = cNumberOfThreads;
4
5        algoRunArgs.numberofthreads_    = numberofthreads_;
```

Listing 5.2: LenaMCRecoStrategy: Speicherung der Anzahl der Threads in den Laufargumenten der Rekonstruktion

Alternativ könnte man es mit `omp_set_dynamic(1)` dem Compiler überlassen, die Anzahl der Threads dynamisch festzulegen, damit zur Laufzeit eine optimale Rescourcenausnutzung stattfindet. Im Abschnitt 6.3 wird dies unter Berücksichtigung der empfohlenden Anzahl an Threads diskutiert.

```
1    size_t n;
2    omp_set_dynamic(0);
3    omp_set_num_threads(numberofthreads);
4    #pragma omp parallel for private(n)\
5    firstprivate(cellArgs) lastprivate(cellArgs)\
6    shared(args, mesh, detEff, unhitPmtIDs, cout, counter)\
7    schedule(guided, 8)
8    for(n = 0; n < nPmts; ++n)
```

Listing 5.3: EmittedLightRecoAlgo: DoRun For-Schleife mit OpenMP

Weiter wird mit dem Befehl `#pragma omp parallel for` die eigentliche Parallelisierung gestartet. Mit dem `private` Befehl wird dem Compiler mitgeteilt, dass für den Schleifenzähler n eine Instanz pro Iteration erzeugt werden soll, jeder Thread also eine eigene Kopie der Variablen in seinem Speicherbereich erstellt, damit sich die Threads nicht gegenseitig beeinflussen. Für die Argumente für die einzelne Zellenberechnung `cellArgs` wird mit dem `firstprivate` Befehl festgelegt, dass ebenfalls jeweils für jede Iteration eine Kopie des Objektes erzeugt wird, hier allerdings mit den außerhalb der Schleife schon initialisierten Exemplarvariablen. Mit dem `lastprivate` Befehl wird zudem festgelegt, dass die Werte der letzten Iteration nach der Schleife in das gemeinsame Objekt übertragen werden [9]. Da auf die weiteren Argumente `args, mesh, detEff, unhitPmtIDs, cout` und `counter` konfliktfrei parallel zugegriffen werden kann, reicht es, wenn für diese insgesamt nur eine Instanz erzeugt wird. Mit dem `schedule(guided, 8)` Befehl wird die dynamische Zuteilung der Schleifeniterationen an die Threads vorgegeben, wobei die Größe der Pakete am

Anfang noch sehr hoch ist und sich dann mit der Zeit auf 8 reduziert.

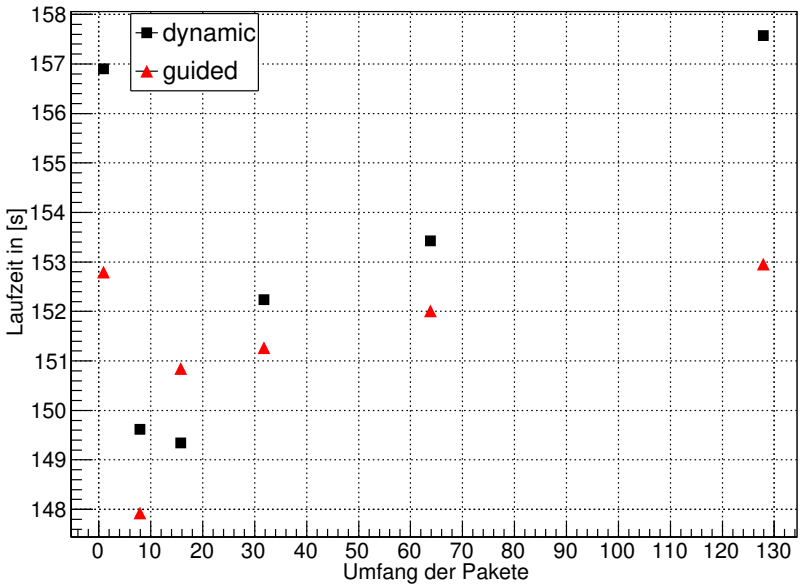

Abbildung 5.2.: Laufzeit in Abhängigkeit vom Zuteilungsverfahren, in schwarz die dynamische Zuweisung mit fester Paketgröße und in rot die dynamische Zuteilung mit kleiner werdener Paketgröße.

Wie in Abbildung 5.2 zu sehen ist, hat sich diese Methode der Zuteilung als am effizientesten herausgestellt. Dabei berechnet sich die Größe der Pakete, die den Threads zugeteilt werden, nach folgender Formel mit dem Anfangswert β_0 für die Anzahl der Schleifeniterationen:

$$\pi_k = \frac{\beta_k}{2N}. \tag{5.1}$$

Dabei ist N die Anzahl der Threads, π_k die Größe des k-ten Paketes und β_k die Anzahl der noch nicht verteilten Schleifeniterationen [1]. Durch diese Verteilung kann eine sehr gleichmäßige Auslastung der einzelnen Rechenkerne erreicht werden, ohne dass ein allzugroßer Aufwand für die Zuordnung der einzelnen Schleifeniterationen entsteht, und so die Rechenzeit weiter reduziert werden.
Die Rekonstruktion speichert in einem C++ Vektor alle PMTs, die nicht getroffen wurden. Da das Anfügen eines Elements an einen C++ Standard Vektor parallel nicht möglich ist, im Gegensatz zu einem indexbasierten Zugriff, wurde der Vektor in der Größe der Anzahl an PMTs bereits vor der For-Schleife erzeugt. Dadurch ist keine weitere Speicherreservierung zur Laufzeit der Schleife nötig. Wie in Listing 5.4 zu sehen, werden alle Vektoreinträge mit 0 initialisiert.

```
1    std::vector<size_t> unhitPmtIDs(nPmts, 0); // use of vector of size
        ↪ nPmts to prevent segmentations breaks
```

Listing 5.4: EmittedLightRecoAlgo::DoRun: vorinitialiserter Vektor

Die Information, ob ein PMT nicht getroffen wurde, wird nun dadurch gespeichert, dass im `else`-Zweig der `if`-Abfrage im Schleifenrumpf der Eintrag an der entsprechenden Stelle auf 1 gesetzt wird:

```
1    unhitPmtIDs[n] = 1;    // remember IDs of unhit PMTs
2    AdaptCellContentArgs(cellArgs,pmt);
3    AddLightDetEffContribution(cellArgs,*detEff);
```

Listing 5.5: EmittedLightRecoAlgo: Speichern nicht getroffener PMTs

Entsprechend muss auch die Methode `EvaluateResults` angepasst und um eine Abfrage auf den Inhalt des Vektoreintrages ergänzt werden (siehe Listing 5.6), sodass der Schleifenkörper mit der Auswertung nur für nicht getroffene PMTs ausgeführt wird.

```
1    if(unhitPmtIDs[iPmtID] == 1)
```

Listing 5.6: EmittedLightRecoAlgo: auslesen nicht getroffener PMTs

Ebenfalls parallelisiert wurde die Schleife über die PMTs in der Methode `EvaluateResults` (siehe Listing 5.7), da sie sich auch als sehr zeitintensiv herausgestellt hat. Aufgrund der geänderten Speicherung der nicht getroffenen PMTs muss die Schleife hier nun über alle PMTs laufen.

```
1    omp_set_dynamic(0);
2    omp_set_num_threads(cellArgs.numberofthreads_);
3    #pragma omp parallel for private(iPmtID)\
4    firstprivate(cellArgs) lastprivate(cellArgs)\
5    shared(unhitPmtIDs, endID, mesh, cout, counter)\
6    schedule(guided, 8)
7    for(iPmtID = 0; iPmtID < endID; ++iPmtID)
```

Listing 5.7: EmittedLightRecoAlgo: EvaluateResults For-Schleife mit OpenMP

Auch hier wurde der Schleifenzähler `iPmtID` privat für jede Iteration gesetzt. Wie auch bei der Schleife im `DoRun`, wurde das Objekt mit den Zellargumenten `cellArgs` als `firstprivate` und `lastprivate` gesetzt. Durch die vorher beschriebene Modifikation des Vektors `unhitPmtIDs` kann dieser nun von allen Threads parallel genutzt werden.

`endID` gibt die Nummer des letzten PMTs an und damit die Größe des Vektors `unhitPmtIDs` und kann von allen Threads parallel genutzt werden, wie auch die Konsolenausgabe `cout`, der Zähler `counter` und das Gitternetz `mesh`. Auch hier findet wieder eine zur Laufzeit dynamische Zuteilung der Schleifeniterationen zu den Threads nach dem zuvor beschriebenen Schema statt.

5.3.3. DetectedLightRecoAlgo

Der Algorithmus zur Rekonstruktion des detektierten Lichtes (siehe Abschnitt 3.2) definiert eine Methode mit einer Schleife über alle PMTs und somit den zeitaufwändigsten Abschnitt des Programmes, der hier ebenfalls parallelisiert wurde.
Der `DetectedLightRecoAlgo` enthält ebenfalls eine `DoRun` Methode mit einer Schleife, die hier parallelisiert wurde (siehe Listing 5.8).

```
1    #pragma omp parallel for private(n)\
2    firstprivate(cellArgs) lastprivate(cellArgs)\
3    shared(args, mesh, cout, counter)\
4    default(none) schedule(guided, 8)
5    for(n = 0; n < nPmts; ++n)
```

Listing 5.8: DetectedLightRecoAlgo: For-Schleife mit OpenMP

Wie auch bei den zuvor beschriebenen Parallelisierungen sind Schleifenzähler und Zellargumente `private`, beziehungsweise `firstprivate` und `lastprivate`, die anderen Variablen werden mit allen Threads gemeinsam genutzt. Auch hier erfolgt die Zuteilung der Schleifeniterationen dynamisch.

5.3.4. RecoAlgorithm

Die beiden zuvor genannten Klassen erben vom `RecoAlgorithm`. Darin wird die Methode `MakePmtSignalFunction` definiert, die nicht ohne Weiteres parallel ausführbar ist, da hier auf ROOT Methoden zurückgegriffen wird, die nicht thread-safe implementiert sind.
Besonders kritisch war hier das parallele Befüllen eines ROOT Histogramms in der aufgerufenen Methode `CalculateWeights`

```
1    PmtSigFcnWeightCalc::CalculateWeights(probMask,
2        *cell_content_provider_,*detector_,cellArgs,*weights);
```

Listing 5.9: RecoAlgorithm: Berechnen des Gewichtungshistogramms

und das abschließende Bauen der `PmtSignalFunction` (siehe Listing 5.10).

```
1    return sfcn_builder_->BuildPmtSignalFunction(pmt.GetChannel(),
2        pmt.GetHits(),*weights);
```

Listing 5.10: RecoAlgorithm: Bauen der PmtSignalFunction

Die erste Schwierigkeit wurde gelöst, indem in der Methode `AddToWeightsTable` die berechneten Wahrscheinlichkeits-Gewichte und Signalzeiten zunächst in vorreservierten Arrays gespeichert werden (siehe Listing 5.12) anstatt das Histogramm direkt zu befüllen. Hier wurden in der Methode `CalculateWeights` je ein Zeit- und Gewichtungsarray für jeden der beiden möglichen Aufrufe von `AddToWeightsTable` erstellt (siehe Listing 5.11), um Überläufe der Arrays zu verhindern.

```
1    size_t ntimes = maskMesh.GetNCells();
2    double * timeArray = new double[ntimes];
3    double * weightArray = new double[ntimes];
4    double * timeArray2 = new double[ntimes];
5    double * weightArray2 = new double[ntimes];
6    unsigned long index = 0;
7    unsigned long index2 = 0;
```

Listing 5.11: PmtSigFcnWeightCalc: Arrays für das Histogramm

```
1    weightArray[index] = lightProb * cellContent * cellVolume;
2    timeArray[index] = sigTime;
```

Listing 5.12: PmtSigFcnWeightCalc: Befüllen der Gewichtungsarrays

Die Arrays werden dann mit Hilfe der `FillN` Methode aus ROOT am Ende der Methode `CalculateWeights` in einer OpenMP-Critical-Umgebung (siehe Listing 5.13), die nur einen Thread zur Zeit in den Codebereich lässt, in das Histogramm gefüllt. Nur so lassen sich Speicherzugriffsfehler an dieser Stelle verhindern:

```
1    #pragma omp critical (Fill)
2    {
3        weights.FillN(index, timeArray, weightArray, 1);
4        weights.FillN(index2, timeArray2, weightArray2, 1);
5    }
```

Listing 5.13: PmtSigFcnWeightCalc: Befüllen des Gewichtungshistogramms

Die zweite kritische Stelle führte ebenfalls zu Speicherzugriffsfehlern; hier beim Erzeugen einer `ROOT TF1` Funktion zum Bauen einer `PmtSignalFunction` in der Methode `BuildFullPmtSignalFunction` der Klasse `ScintillatorPmtSignalFunctionBuilder`. Um diesen Fehler zu beheben, wurde ein `omp critical`-Block um die Erzeugung des `boost::scoped_ptr<TF1> componentFunction` und die anschließende Schleife über alle Treffer auf den jeweiligen PMT gesetzt (siehe Listing 5.14).

```
1   #pragma omp critical (TF1)
2   {
3   // create ROOT TF1 function for component to full PMT signal
        ↪ function
4   boost::scoped_ptr<TF1> componentFunction(new TF1("comp_tf1",this,
5           &ScintillatorPmtSignalFunctionBuilder::FullSignalComponent,
6           lower_edge_,upper_edge_,2));
7   componentFunction->SetNpx(num_bins_);
8
9   // loop over all hits contributing to the full signal histogram
10  DetEvt::Hits::const_iterator iHit = hits.begin();
11  for(; iHit != hits.end(); ++iHit)
12  {
13      [...]
14  }
15  }
```

Listing 5.14: ScintillatorPmtSignalFunctionBuilder: OMP Critical Block

Dies führt allerdings dazu, dass immer nur eine Signalfunktion pro Zeit berechnet werden kann, sodass an dieser Stelle das Potenzial der Parallelisierung nicht genutzt wird.

6. Auswertung

In diesem Kapitel wird zunächst die Testumgebung beschrieben und anschließend werden die Messungen der Rechenzeiten analysiert und bewertet.

6.1. Testumgebung

Die Rekonstruktion wurde auf einem Workgroup Server mit 2 Intel Xeon X5650 CPUs, die jeweils mit 6 Kernen ausgestattet sind und softwareseitig mit gcc 5.4.0, ROOT 6.04.18, Boost 1.61.0 und CLHEP 2.3.3.1 getestet.
Die Kerne unterstützen die Intel Hyper-Threading-Technik, mit der auf jedem physikalischen Kern zwei Threads parallel laufen können [10]. Da zum Zeitpunkt dieser Arbeit das Programm der topologischen Rekonstruktion noch nicht an den JUNO Dektektor angepasst wurde, wurden die Testereignisse basierend auf den Planungen für den zylindrischen Flüssigszintillatordetektor LENA simuliert und rekonstruiert.

6.2. Leistungsanalyse

Um den Erfolg der Parallelisierung zu analysieren, wurden verschiedene Messreihen gestartet. Hierfür wurden zwei verschiedene Ereignisse benutzt. Zum einen wurde ein Elektronereignis mit einer Energie von 5 MeV, das in der Mitte des Detektors ein Signal erzeugt, rekonstruiert. Dies entspricht in etwa der Energie, die man für ein durch Neutrions hervorgerufenes Ereignis im einfachsten Fall erwartet. Als zweites Ereignis wurde ein Myon mit einer Energie von 40 GeV, das diagonal durch den Detektor fliegt, verwendet. Dies entspricht einem Untergrundereignis, wie man es im ungünstigstem Fall erwartet.

In Abbildung 6.1 ist die Laufzeit für eine Iteration mit 100 cm Gitterweite für das Elektronereignis zu sehen. Deutlich erkennbar ist, dass die Laufzeit mit höherer Threadanzahl,

gegen Ende schwächer, abnimmt. Die durchgeführte Parallelisierung ist im Messbereich also unabhängig von der Threadanzahl erfolgreich. Wie in Abbildung 6.2 zu sehen ist, lässt sich allerdings in der Beschleunigung ab 12 Threads ein Knick feststellen, zudem erhöht sich der Beschleunigungsfaktor nur bei geraden Threadanzahlen. Die maximale Beschleunigung liegt bei 22 Threads mit 12,49 vor.

Die Laufzeiten für eine Iteration für das Myonereignis sind in Abbildung 6.3 dargestellt. Im Vergleich zum Elektronereignis fällt auf, dass auch hier die Laufzeit zunächst starkt abfällt, dann aber ein Minimum zu erreichen scheint. In Abbildung 6.4 ist zu sehen, dass der Beschleunigungsfaktor ab 12 Threads etwa konstant bei 5,6 bleibt.

Führt man die Messung für das Elektronereignis mit acht Iterationen aus (Konfiguration siehe Listing B.2), so bestätigen sich die Beobachtungen, die auch für eine einzelne Iteration gemacht wurden: In Abbildung 6.6 ist der Knick in der Beschleunigung zwischen 12 und 13 Threads noch deutlicher zu sehen. Aufgrund der großen Zeitskala in Abbildung 6.5 sind die Laufzeitschwankungen nur schwach zu erkennen. Der maximale Beschleunigungsfaktor beträgt hier 12,73.

Wie in Abbildung 6.7 zu sehen ist, ergibt sich beim Myonereignis auch bei vier Iterationen mit je 100 cm Gitterweite (Konfiguration siehe Listing B.1) fast kein Laufzeitvorteil durch die Verwendung von mehr als 12 Threads. In Abbildung 6.8 kann man sehen, dass die Beschleunigung ihr Maximum bei etwa 12 Threads mit einem Faktor von 5,4 erreicht.

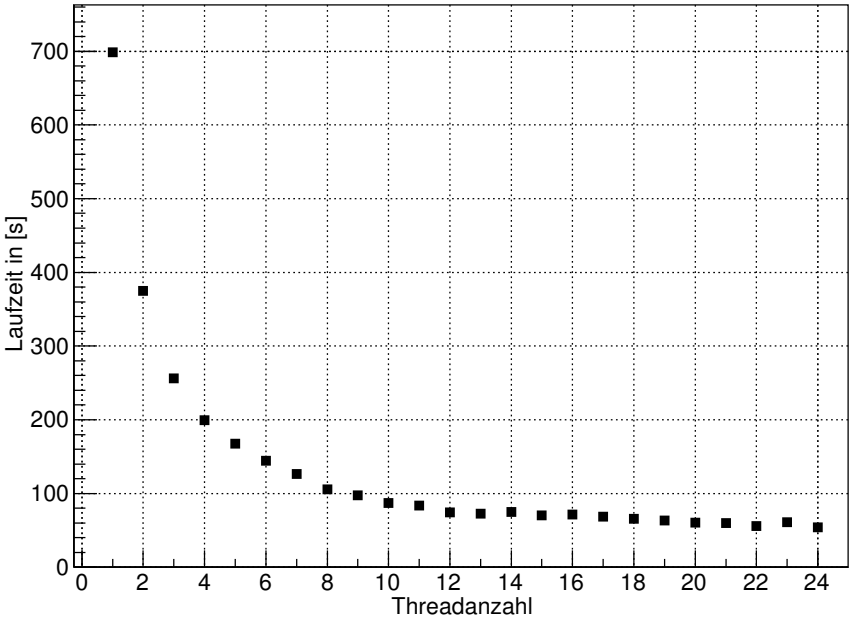

Abbildung 6.1.: Laufzeit einer Iteration der Rekonstruktion für ein Elektronereignis in Abhängigkeit von der Anzahl der Threads.

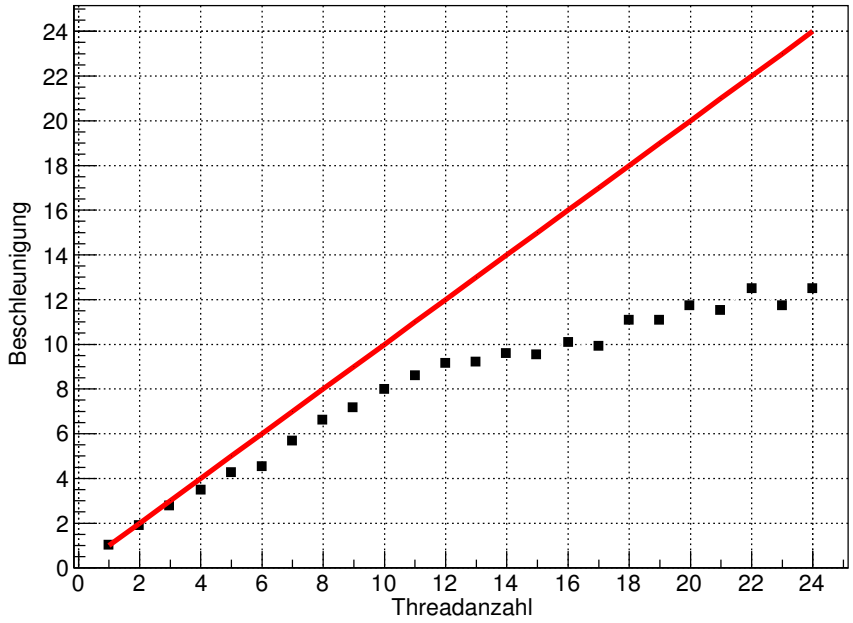

Abbildung 6.2.: Beschleunigung (schwarze Vierecke) einer Iteration der Rekonstruktion für ein Elektronereignis in Abhängigkeit von der Anzahl der Threads. In rot ist die ideale Beschleunigung eingezeichnet.

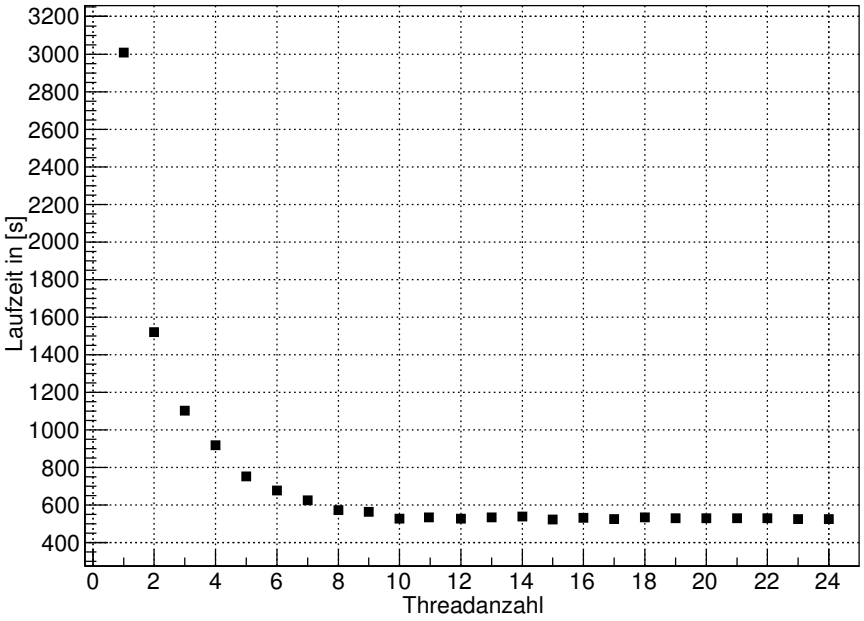

Abbildung 6.3.: Laufzeit einer Iteration der Rekonstruktion für ein Myonereignis in Abhängigkeit von der Anzahl der Threads.

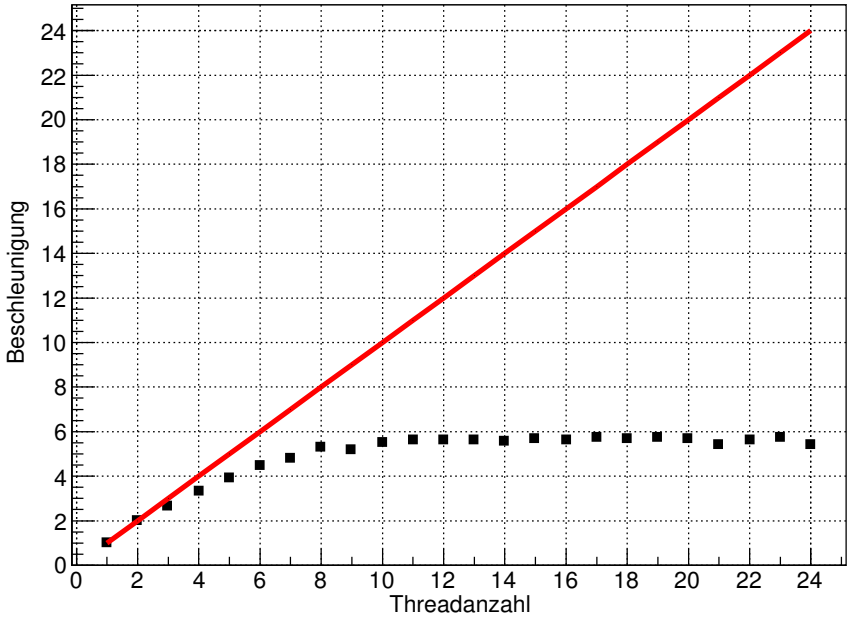

Abbildung 6.4.: Beschleunigung (schwarze Vierecke) einer Iteration der Rekonstruktion für ein Myonereignis in Abhängigkeit von der Anzahl der Threads. In rot ist die ideale Beschleunigung eingezeichnet.

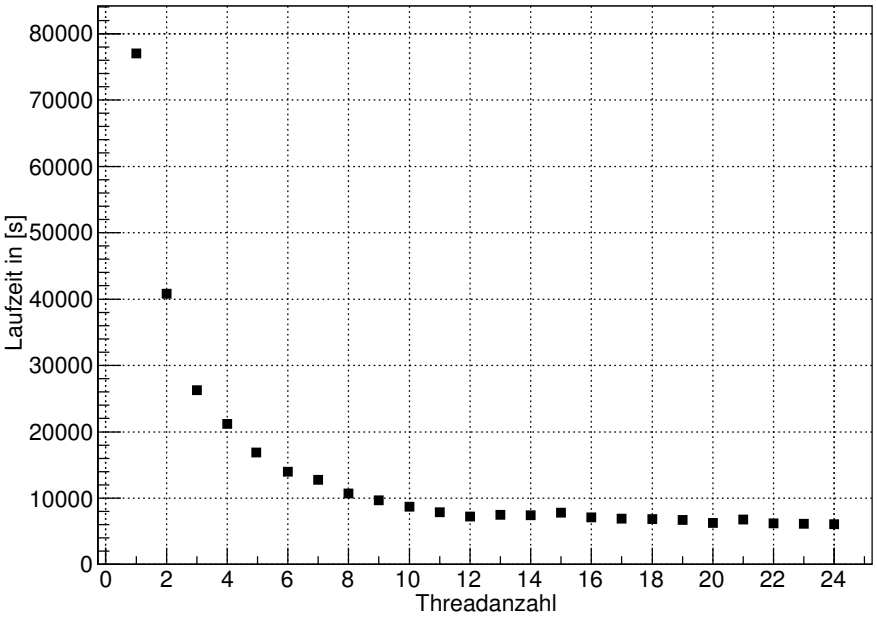

Abbildung 6.5.: Laufzeit von acht Iterationen der Rekonstruktion für ein Elektronereignis in Abhängigkeit von der Anzahl der Threads.

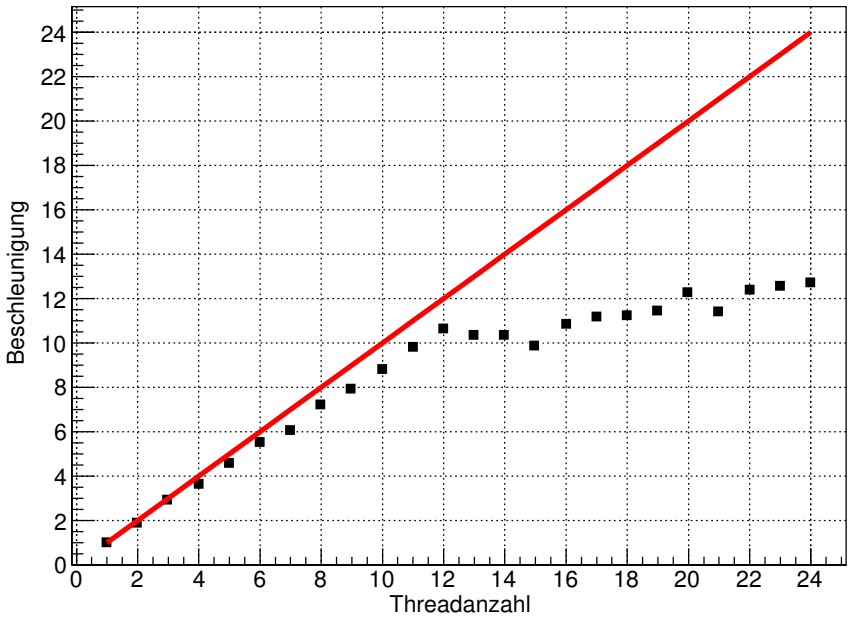

Abbildung 6.6.: Beschleunigung (schwarze Vierecke) von acht Iterationen der Rekonstruktion für ein Elektronereignis in Abhängigkeit von der Anzahl der Threads. In rot ist die ideale Beschleunigung eingezeichnet.

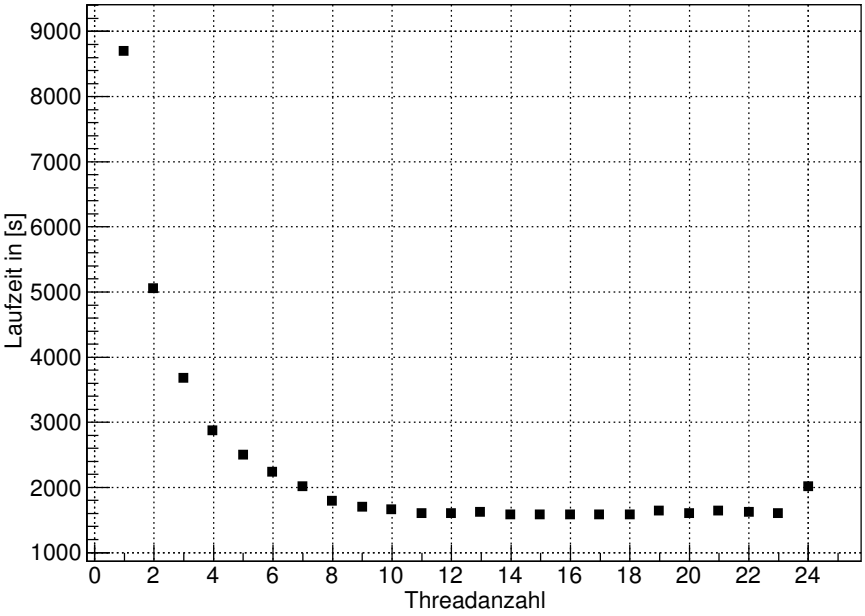

Abbildung 6.7.: Laufzeit von vier Iterationen der Rekonstruktion für ein Myonereignis in Abhängigkeit von der Anzahl der Threads.

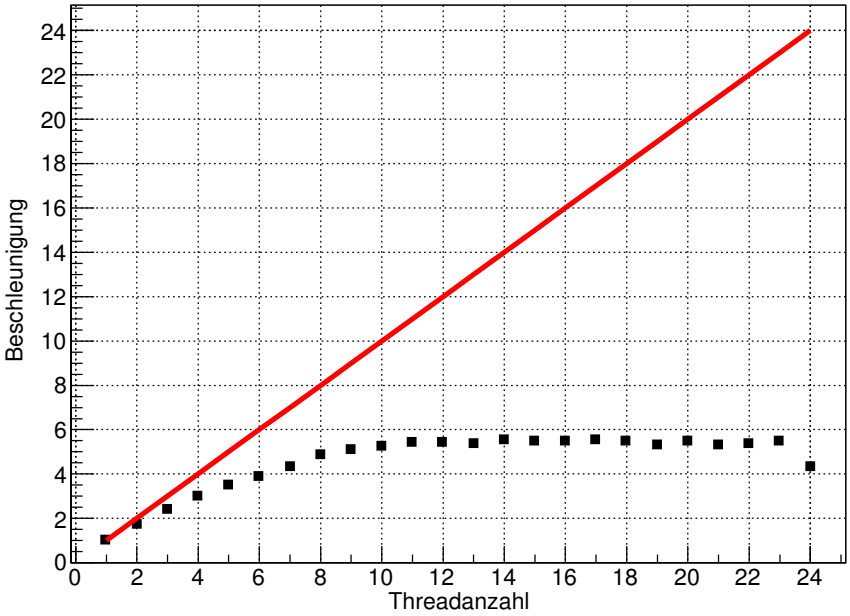

Abbildung 6.8.: Beschleunigung (schwarze Vierecke) von vier Iterationen der Rekonstruktion für ein Myonereignis in Abhängigkeit von der Anzahl der Threads. In rot ist die ideale Beschleunigung eingezeichnet.

Parallele Ausführung mehrerer Rekonstruktionen

Zur Laufzeit der Messungen wurden keine anderen Programme auf dem Workgroup Server ausgeführt. In weiteren Messungen mit dem Elektronereignis (siehe Tabelle 6.1 und Tabelle 6.2) sieht man aber, dass bei gleichzeitiger Ausführung mehrerer Rekonstruktionen, die zusammen mehr als 12 Kerne belegen, deutliche Laufzeitverlängerungen auftreten. Werden vier Rekonstruktionen mit jeweils 6 Threads gleichzeitig ausgeführt, so dauert jede einzelne Rekonstruktion im Durchschnitt 45,44% länger, als wenn sie als einzigstes Programm auf dem Rechner laufen würde. Bei 2 Rekonstruktionen mit je 12 Threads verlängert sich die Laufzeit einer Rekonstruktion durchschnittlich um 51,66%.

	1 x 6 Threads	2 x 6 Threads	4 x 6 Threads
Laufzeit 1 [s]	144,672	144,569	195,024
Laufzeit 2 [s]		151,204	210,319
Laufzeit 3 [s]		147,56	214,14
Laufzeit 4 [s]		148,467	222,179
Durchschnitt [s]	144,672	147,950	210,416
Veränderung zu 1 x 6	0,0%	2,27%	45,44%

Tabelle 6.1.: Laufzeit einer Rekonstruktion bei parallelem Lauf mehrerer Rekonstruktionen mit 6 Threads

	1 x 12 Threads	2 x 12 Threads
Laufzeit 1 [s]	74,2037	107,041
Laufzeit 2 [s]		120,937
Laufzeit 3 [s]		111,06
Laufzeit 4 [s]		111,118
Durchschnitt [s]	74,204	112,539
Veränderung zu 1 x 12	0,0%	51,66%

Tabelle 6.2.: Laufzeit einer Rekonstruktion bei parallelem Lauf mehrerer Rekonstruktionen mit 12 Threads

6.3. Bewertung

Den Knick bei jeweils 12 Threads kann man auf die Prozessorarchitektur zurückführen. Bis 12 Threads wird jeder Thread auf einem eigenen Kern gerechnet, danach wird das Hyper-Threading verwendet, sodass in einem Kern zwei Threads gerechnet werden. Der Hyperthreading-Kern ist dabei allerdings nicht so leistungsfähig wie ein vollwertiger Prozessorkern.

Wie bei der Leistungsanalyse zu sehen ist, hängt die maximale Beschleunigung auch vom Ereignis ab. Bei einer Iteration erreicht man bei 12 Threads auf dieser Hardware beim Elektronereignis eine Beschleunigung von 9,1, für das Myonereignis nur von 5,6. Für Ereignisse mit größerem Energieverlust im Detektor, d.h. für Myonereignisse, ist die Parallelisierungseffizienz schlechter, da wesentlich mehr PMTs getroffen werden und die Methoden zur Berechnung eines getroffenen PMTs nicht vollständig parallel ausführbar sind. Es kommt dadurch zu Wartezeit der Threads an den nicht-parallelen Code-Bereichen (siehe Abschnitt 5.3).
Wie in Abbildung 6.3 und Abbildung 6.7 zu sehen ist, wird für Myonereignisse bei 12 Threads ein Minimum der Laufzeit erreicht. Somit ist eine weitere Erhöhung der Threadanzahl bei dieser Hardware nicht sinnvoll. Daher sollte die Threadanzahl ereignisspezifisch gewählt werden und nicht unbedingt der Prozessorkernanzahl entsprechen. Es kann aber durchaus sinnvoll sein, wie im Abschnitt 6.2 beschrieben, die zur Verfügung stehenenden Prozessorkerne auf mehrere seperate Prozesse aufzuteilen, wenn mehrere Ereignisse in insgesamt kürzerer Zeit berechnet werden sollen. Möchte man hingegen das Ergebnis für ein Ereignis möglichst schnell berechnen, so sollte man nur eine Rekonstruktion zur Zeit ausführen.

7. Zusammenfassung

Die topologische Rekonstruktion der Spurdaten von Ereignissen in einem Flüssigszintillatordetektor ermöglicht zum einen die Identifizierung der Teilchen im Detektor, zum anderen das Festlegen eines Vetovolumens rund um durch Untergrund erzeugte Ereignisse. Ziel dieser Arbeit war es, das Potenzial einer Parallelisierung zu untersuchen und die Rechenzeit der Rekonstruktion unter der Ausnutzung der vorhandenen Rechenkapazitäten zu reduzieren. Dies ist mit der Parallelisierung mittels OpenMP gut gelungen. Dafür wurden die zeitaufwändigsten Schleifen auf mehrere Threads aufgeteilt und auf mögliche Konflikte geachtet. Die Berechnung für die einzelnen PMTs findet nun parallel statt. Es hat sich gezeigt, dass eine dynamische Zuteilung der PMTs zu den Threads mit zunächst sehr großer Paketgröße, die dann kleiner wird, am sinnvollsten ist, da die Rechenzeit für jeden PMT variiert. Die zwei Konfliktstellen des Programmes, die eine Parallelisierung erschwerten, konnten lokalisiert werden. Das Erstellen der Histogramme für den Bau der Signalfunktion konnte angepasst werden. Das Bauen der Signalfunktion selbst muss aber noch verändert werden, um auch hier eine konfliktfreie parallele Ausführung zu ermöglichen.
Für alle untersuchten Ereignisse im Detektor konnte eine deutliche Reduzierung der Rechenzeit erreicht werden, bei Einsatz aller 24 Prozessorkerne der Testumgebung konnte eine maximale Beschleunigung von 12,7 für das 5 MeV Elektronereignis und von 5,72 für das 40 GeV Myonereignis erzielt werden.

7.1. Fazit und Ausblick

Für die verwendete Hardware wurde ein sehr gutes Ergebnis erreicht. Allerdings muss der Algorithmus selbst noch angepasst werden, um noch schneller Ergebnisse zu bekommen. Gerade für die durch Myonen erzeugten Ereignisse gibt es noch einigen Handlungsbedarf, um die nicht-parallelen Programmabschnitte zu verkürzen. Hier sollte noch untersucht

werden, ob es eine parallel aufrufbare Methode zum Bauen der Signalfunktion gibt. Auch eine Optimierung der zuberechnenden Detektorbereiche sollte geprüft werden, um so die Zahl der nötigen Berechnungen zu reduzieren.
Insgesamt wurde aber ein wichtiger Schritt Richtung Echtzeitrekonstion gemacht, da die einzelnen Rechungen in Sekundenbruchteilen gerechnet werden können und somit eine gute Skalierbarkeit der Parallelisierung angenommen werden kann. Damit wurden auch die strukturellen Grundlagen geschaffen, die eine Parallelisierung im größeren Maßstab ermöglichen. Die Parallelisierung hat sich für die topologische Rekonstruktion als ein sehr sinnvolles Instrument herausgestellt.

7.1.1. Empfehlung für Hardware

Es hat sich gezeigt, dass die Leistung der Parallelisierung nach erreichen der Anzahl an physikalischen Prozessorkernen durch Hyperthreading nicht im gleichen Maß weiter gesteigert werden kann. Daher sollte bei der Hardware darauf geachtet werden, dass möglichst viele physikalische Prozessorkerne zur Verfügung stehen, da die Parallelisierung gut mit der Prozessorkernanzahl skaliert. Bei der Rekonstruktion von myoninduzierten Ereignissen wird dies allerdings erst entscheidend, wenn die Parallelisierung nicht mehr so stark durch die nicht-parallelisierbaren Abschnitte begrenzt ist. Zudem kann die Rekonstruktion sehr speicherintensiv sein, sodass ausreichend Arbeitsspeicher vorgehalten werden sollten.
Aktuell wird über den Einsatz von Grafikkarten für die weitere Beschleunigung der topologischen Rekonstruktion nachgedacht. Grafikkarten sind besonders für aufwendige Matrix- und Vektoroperationen geeignet, die aktuell nicht in der Rekonstruktion enthalten sind. Daher muss die Rekonstruktion dafür stark angepasst werden. Ob sich damit die Berechnungen der Rekonstruktion beschleunigen lassen, wurde in dieser Arbeit nicht untersucht.

A. Laufzeitmessungen

Nachfolgend finden sich die Zeiten der Laufzeitmessungen, die im Rahmen dieser Arbeit durchgeführt wurden.

# Threads	Laufzeit	Beschleunigung
1	790,494	1
2	407,232	1,941
3	299,369	2,641
4	232,106	3,406
5	194,145	4,072
6	177,932	4,443
7	165,592	4,774
8	147,338	5,365
9	135,572	5,831
10	127,352	6,207
11	126,898	6,229
12	122,019	6,478
16	113,09	6,99
20	104,993	7,529
22	101,307	7,803

Tabelle A.1.: Laufzeiten und Beschleunigung zur Parallelisierung mit Boost::Threads: In Abbildung 5.1 ist die Beschleunigung mit Boost::Threads aufgetragen

Größe der Blöcke	Laufzeit dynamic	Laufzeit guided
1	156,901	152,777
8	149,619	147,927
16	149,327	150,827
32	152,221	151,263
64	153,415	151,996
128	157,574	152,945

Tabelle A.2.: Laufzeiten in Abhängigkeit vom Zuteilungsverfahren: In Abbildung 5.2 sind die Laufzeiten für eine Interation mit einem Elektronereignis für unterschiedliche Zuteilungsverfahren aufgetragen

# Threads	Laufzeit Run 1	Laufzeit Run 2	Laufzeit Run 3	∅ Laufzeit	Beschleunigung
1	698,422	713,085	692,811	701,4393	1,0000
2	374,651	371,913	376,496	374,3533	1,8737
3	254,787	256,417	250,928	254,0440	2,7611
4	199,289	201,411	202,231	200,9770	3,4901
5	167,524	160,214	164,742	164,1600	4,2729
6	144,672	161,344	158,23	154,7487	4,5328
7	126,379	122,022	121,818	123,4063	5,6840
8	105,585	106,298	107,115	106,3327	6,5966
9	97,7464	97,99	96,9916	97,5760	7,1886
10	86,9436	88,5796	87,5048	87,6760	8,0004
11	83,5072	79,9262	80,91	81,4478	8,6121
12	74,2037	82,7587	73,0433	76,6686	9,1490
13	72,4945	76,1071	79,6853	76,0956	9,2179
14	75,0605	75,1153	69,3757	73,1838	9,5846
15	70,0746	71,4083	78,8291	73,4373	9,5515
16	71,1375	71,4936	66,6556	69,7622	10,0547
17	68,3372	73,1059	70,6088	70,6840	9,9236
18	65,3486	61,8068	62,9714	63,3756	11,0680
19	63,3908	64,5372	62,5502	63,4927	11,0476
20	60,3771	59,5257	59,2937	59,7322	11,7431
21	59,9468	59,4857	63,5046	60,9790	11,5030
22	55,9098	55,9448	56,6632	56,1726	12,4872
23	61,1442	58,9174	59,4559	59,8392	11,7221
24	53,7687	57,1576	57,6955	56,2073	12,4795

Tabelle A.3.: Laufzeiten und Beschleunigung von einer Iteration mit einem Elektronereignis: In Abbildung 6.1 ist Run 1 und in Abbildung 6.2 die Beschleunigung der durchschnittlichen Laufzeit aufgetragen

# Threads	Laufzeit Run 1	Laufzeit Run 2	Laufzeit Run 3	∅ Laufzeit	Beschleunigung
1	2888,25	3009,12	3114,23	3003,8667	1,0000
2	1482,53	1519,83	1479,55	1493,9700	2,0107
3	1186,75	1101,47	1079,03	1122,4167	2,6762
4	917,452	917,588	890,1	908,3800	3,3068
5	761,979	753,528	787,296	767,6010	3,9133
6	669,138	677,238	671,617	672,6643	4,4656
7	624,311	625,267	619,539	623,0390	4,8213
8	573,839	572,851	560,236	568,9753	5,2794
9	632,984	563,097	540,014	578,6983	5,1907
10	549,845	528,635	555,343	544,6077	5,5157
11	539,034	530,637	540,748	536,8063	5,5958
12	532,266	528,384	535,158	531,9360	5,6470
13	533,413	534,511	537,677	535,2003	5,6126
14	537,818	539,605	537,831	538,4180	5,5791
15	524,744	524,455	533,853	527,6840	5,6925
16	532,886	531,362	531,627	531,9583	5,6468
17	524,782	526,096	526,08	525,6527	5,7145
18	524,295	535,187	533,959	531,1470	5,6554
19	523,698	530,976	521,145	525,2730	5,7187
20	523,936	530,481	526,808	527,0750	5,6991
21	619,935	530,731	518,613	556,4263	5,3985
22	538,423	530,213	531,34	533,3253	5,6323
23	522,27	525,497	530,337	526,0347	5,7104
24	523,16	524,683	617,73	555,1910	5,4105

Tabelle A.4.: Laufzeiten und Beschleunigung von einer Iteration mit einem Myonereignis: In Abbildung 6.3 ist Run 2 und in Abbildung 6.4 die Beschleunigung der durchschnittlichen Laufzeit aufgetragen

# Threads	Laufzeit	Beschleunigung
1	77039,6	1,00
2	40780,8	1,89
3	26249,1	2,93
4	21218,8	3,63
5	16771,5	4,59
6	13981,7	5,51
7	12754,5	6,04
8	10676,4	7,22
9	9699,75	7,94
10	8717,89	8,84
11	7849,35	9,81
12	7250,54	10,63
13	7462,48	10,32
14	7440,85	10,35
15	7825,12	9,85
16	7117,84	10,82
17	6904,95	11,16
18	6864,63	11,22
19	6730,49	11,45
20	6265,2	12,30
21	6764,02	11,39
22	6231,5	12,36
23	6137,72	12,55
24	6051,71	12,73

Tabelle A.5.: Laufzeiten und Beschleunigung von acht Iterationen mit einem Elektronereignis: In Abbildung 6.5 ist Laufzeit und in Abbildung 6.6 die Beschleunigung der Laufzeit aufgetragen

# Threads	Laufzeit	Beschleunigung
1	8695,02	1,00
2	5044,43	1,72
3	3661,67	2,37
4	2874,88	3,02
5	2494	3,49
6	2234,31	3,89
7	2014,76	4,32
8	1784,85	4,87
9	1700,56	5,11
10	1658,23	5,24
11	1599,05	5,44
12	1602,85	5,42
13	1622,13	5,36
14	1577,23	5,51
15	1588,58	5,47
16	1582,61	5,49
17	1578,6	5,51
18	1586,36	5,48
19	1629,64	5,34
20	1591,57	5,46
21	1633,46	5,32
22	1626,42	5,35
23	1593,04	5,46
24	2003,38	4,34

Tabelle A.6.: Laufzeiten und Beschleunigung von vier Iterationen mit einem Myonereignis: In Abbildung 6.7 ist die Laufzeit und in Abbildung 6.8 die Beschleunigung der Laufzeit aufgetragen

B. Konfigurationsdateien

Nachfolgend finden sich Ausschnitte aus zwei der verwendeten Konfigurationsdateien, die die Anzahl und Gitterweite der durchgeführten Iterationen, sowie die Anfangswerte für den Referenzpunkt und die Referenzzeit definieren.

```
1  # Strategy configuration
2  [Strategy]
3
4  #------------------------------------------------------------------
5  # Add reconstruction iterations to the event reconstruction process.
6
7      Iteration               =   100.0   #cm
8      Iteration               =   100.0   #cm
9      Iteration               =   100.0   #cm
10     Iteration               =   100.0   #cm
11
12 #------------------------------------------------------------------
13 # Smearing of MC truth primary vertex position and time to get reference
14 # parameters for the Wonsak reconstruction
15
16      RefPointSmear.Seed     =   1
17      RefPointSmear.SigmaX   =   10.0   #cm
18      RefPointSmear.SigmaY   =   10.0   #cm
19      RefPointSmear.SigmaZ   =   10.0   #cm
20
21      RefTimeSmear.Seed      =   1
22      RefTimeSmear.SigmaT    =   1.0    #ns
```

Listing B.1: Ausschnitt aus der Konfigurationsdatei für 4 Iterationen

```
# Strategy configuration
[Strategy]

#------------------------------------------------------------------
# Add reconstruction iterations to the event reconstruction process.

    Iteration               =   100.0   #cm
    Iteration               =   100.0   #cm
    Iteration               =   100.0   #cm
    Iteration               =   100.0   #cm
    Iteration               =   100.0   #cm
    Iteration               =    50.0   #cm
    Iteration               =    50.0   #cm
    Iteration               =    12.5   #cm

#------------------------------------------------------------------
# Smearing of MC truth primary vertex position and time to get reference
# parameters for the Wonsak reconstruction

     RefPointSmear.Seed     =   1
     RefPointSmear.SigmaX   =   10.0   #cm
     RefPointSmear.SigmaY   =   10.0   #cm
     RefPointSmear.SigmaZ   =   10.0   #cm

     RefTimeSmear.Seed      =   1
     RefTimeSmear.SigmaT    =   1.0    #ns
```

Listing B.2: Ausschnitt aus der Konfigurationsdatei für 8 Iterationen

Literaturverzeichnis

[1] Akhter, Shameem ; Roberts, Jason: *Multicore Programmierung - Performance erhöhen durch Software-Multithreading*. Entwickler.press, München, 2008. – ISBN 978-3-939084-70-9

[2] An, Fengpeng ; An, Guangpeng ; An, Qi ; Antonelli, Vito ; Baussan, Eric u. a.: Neutrino Physics with JUNO. (2015)

[3] Brands, Gilbert: *Das C++ Kompendium*. 2. Auflage, Teil 2. eXamen.press, Springer-Verlag Berlin Heidelberg, 2010. – ISBN 978-3-642-04786-2

[4] Collaboration, The I.: *Physics Potential of the ICAL detector at the India-based Neutrino Observatory (INO)*. 2015

[5] Collaboration, The IceCube-PINGU: *Letter of Intent: The Precision IceCube Next Generation Upgrade (PINGU)*. 2014

[6] Demtröder, Wolfgang: *Experimentalphysik 4: Kern-, Teilchen- und Astrophysik*. 4. Auflage. Springer Spektrum, Springer-Verlag Berlin Heidelberg, 2014. – ISBN 978-3-642-21475-2

[7] Forschungsgruppe Neutrinophysik Universität Hamburg: *JUNO*. 2016. – URL `http://www.neutrino.uni-hamburg.de/projekte/juno/`. – Zugriffsdatum: 13.09.2016

[8] Group, Particle D.: *NEUTRINO MASS, MIXING, AND OSCILLATIONS*. 2016. – URL `http://pdg.lbl.gov/2016/reviews/rpp2016-rev-neutrino-mixing.pdf`. – Zugriffsdatum: 03.11.2016

[9] Hoffmann, Simon ; Lienhart, Rainer: *OpenMP - Eine Einführung in die parallele Programmierung mit C/C++*. Springer-Verlag Berlin Heidelberg, 2008. – ISBN 978-3-540-73123-8

[10] Intel Corporation: *Intel® Xeon® Processor X5650*. 2016. – URL `http://ark.intel.com/de/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI`. – Zugriffsdatum: 19.09.2016

[11] Johannes Gutenberg Universität Mainz: *Neutrino mass hierarchy in JUNO*. 2016. – URL `http://www.staff.uni-mainz.de/wurmm/juno.html`. – Zugriffsdatum: 13.09.2016

[12] Li, Yu-Feng for the JUNO collaboration: *Jiangmen Underground Neutrino Observatory: Status and Prospectives*. 2016

[13] Lorenz, Sebastian, JGU Mainz: Overview of Muon Tracking Methods, 2016

[14] OpenMP Architecture Review Board: *OpenMP Application Program Interface - Version 4.0*. 2013. – URL `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`. – Zugriffsdatum: 22.08.2016

[15] Pauli, Wolfgang: *Liebe Radioaktive Damen und Herren - Offener Brief an die Gruppe der Radioaktiven bei der Gauvereins-Tagung zu Tübingen*. 1930

[16] Povh, Bogdan ; Rith, Klaus ; Scholz, Christoph ; Zetsche, Frank: *Teilchen und Kerne*. Springer-Verlag Berlin Heidelberg, 2009. – ISBN 978-3-540-68075-8

[17] Wikipedia: *Szintillator — Wikipedia, Die freie Enzyklopädie*. 2016. – URL `https://de.wikipedia.org/w/index.php?title=Szintillator&oldid=159083764`. – Zugriffsdatum: 08.11.2016. – [Online; Stand 8. November 2016]

[18] Zuber, Kai: *Neutrino Physics*. Institute of Physics Publishing, Bristol and Philadelphia, 2004. – ISBN 978-0-7503-0750-1

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

Danksagung

Mein Dank gilt allen, die an der Entstehung dieser Arbeit mitgewirkt haben. Frau Prof. Dr. Caren Hagner für die Möglichkeit diese Arbeit in ihrer Arbeitsgruppe zu schreiben. Dr. Björn Wonsak für das interessante und herausfordernde Thema. Der gesamten Forschungsgruppe Neutrinophysik möchte ich für die netten Gespräche beim gemeinsamen Mittagessen und die Kickerrunden danach danken, so ein gutes Training wird mir sicher mal helfen. Daniel Hartwig danke ich dafür, dass ich sein Programm zur Visualiserung des Rekonstruktionsergebnisses nutzen konnte, um damit die Ergebnisse vor und nach meiner Parallelisierung vergleichen zu können. Vielen Dank auch an Sebastian Lorenz, der mir oft helfen konnte, wenn ich mal nicht mit dem Programm zurecht kam. Benedict Schacht, Daniel Hartwig und Felix Benckwitz danke ich für die netten Stunden im Büro, in denen wir uns gegenseitig immer wieder helfen konnten. Bei David Meyhöfer möchte ich mich für die Anregungen für die Fußnoten bedanken. Vielen Dank auch an Henning Rebber, Felix Benckwitz, Benedict Schacht und Christopher Hahn für die Suche nach Fehlern und Unklarheiten in meiner Arbeit und die zahlreichen Anregungen und Tipps.
Selbstverständlich gilt ein besonderer Dank meiner Familie. Meinen Eltern für die Suche nach Rechtschreibfehlern und dem ein oder anderen fehlenden Komma. Meiner Schwester Friederike möchte ich dafür danken, dass sie mir oft Fragen zur deutschen Sprache kompetent beantworten konnte. Bei Sebastian Döbel möchte ich mich für die LaTeX Vorlage und die Hinweise zur Komplierung von gcc 5.4.0 mittels enviroment modules bedanken. Schade, dass du jetzt nach Dresden abhaust, um dort den Master Informatik zu studieren. Aber heute ist nicht alle Tage, du kommst wieder, keine Frage.Vielen Dank an Melf Johannsen für die Hilfe mit der ein oder anderen Mathe- und Physikaufgabe, ohne die ich vermutlich jetzt noch nicht diesen Text hier schreiben würde. Bei Melf und Sebastian möchte ich mich auch für die netten Grill- und Wasserparties im Stadtpark, die Glühweinverkostungen und die netten Lernrunden vor jeder Mathe- und Physikklausur bedanken.
Vielen Dank an meine Kommilitonen Laura, Melf, Sebastian und Tim für die letzten 3 Jahre, in denen wir gemeinsam die Tücken von Computing in Science überwunden haben. Genauso danke ich natürlich all denen, die hier stehen sollten, aber es auf mysteriöse Weise nicht tun.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Computing in Science, Schwerpunkt Physik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin damit einverstanden, dass meine Abschlussarbeit in den Bestand der Fachbereichsbibliothek und online auf der Website der Arbeitsgruppe Neutrinophysik eingestellt wird.

Hamburg, 10.11.2016

Ort, Datum	Unterschrift