Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Leistungsverbesserung der Simulation von 3D Strahlung auf extrasolare Planeten

vorgelegt von

Oliver Pola

Fakultät für Mathematik, Informatik und Naturwissenschaften

Fachbereich Informatik

Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Computing in Science
Matrikelnummer:	6769946
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Prof. Dr. Peter Hauschildt
Betreuer:	Prof. Dr. Peter Hauschildt, Dr. Michael Kuhn, Jannek Squar

Hamburg, den 06.08.2018

Abstract

PHOENIX ist ein moderner Code zur Modellrechnung von Stern- und Planeten-Atmosphären und kommt bei der Charakterisierung extrasolarer Planeten zum Einsatz. Die fortschreitende Beobachtungstechnik erfordert immer komplexere Modelle, damit einher gehen hohe Laufzeiten zur Berechnung, selbst bei Verwendung von Hochleistungsrechnern und entsprechenden Parallelisierungs-Methoden wie MPI und OpenMP. Diese Arbeit soll daher zur Verbesserung der Leistung von PHOENIX beitragen, wobei die Betrachtungen auf einen Programmteil, den sphärischen Tracker, konzentriert werden.

Zunächst wird dazu eine umfangreiche Leistungsanalyse von PHOENIX vorgenommen, die Ansätze für mögliche Verbesserungen liefert. Dabei werden die Skalierung anhand des Speedups betrachtet, ein Profil erstellt und Spurdaten analysiert.

Die Ergebnisse der Leistungsanalyse legen nahe, sich mit OpenMP zu beschäftigen. Dazu wird das vorhandene OpenMP-Schema anhand von gängigen Vorgehensweisen zur Optimierung von OpenMP-Code überarbeitet. Dadurch lässt sich aber kein Leistungsgewinn erzielen.

Stattdessen konzentriert sich diese Arbeit auf Interna von PHOENIX. Sogenannte Charakteristiken durchlaufen die Elemente des sphärischen Gitters, durch das die Planeten-Atmosphäre dargestellt wird. Um alle Gitterelemente zu erfassen, wird eine unterschiedliche Anzahl an Charakteristiken benötigt. Die Ursachen und Auswirkungen dieser schwankenden Anzahl werden untersucht. Es stellt sich heraus, dass diese Anzahl ein Maß für den Workload des betrachteten Trackers darstellt, weshalb nach Methoden gesucht wird, diese Zahl zu reduzieren. Variationen der verschachtelten Schleifen, in denen die Charakteristiken erzeugt werden, liefern solche Methoden und es zeigt sich, dass zusammen mit der Anzahl der Charakteristiken auch die Laufzeit reduziert werden kann.

Inhaltsverzeichnis

1. Einleitung

Lange Zeit ist man zwar davon ausgegangen, dass unser Sonnensystem nicht einzigartig ist und es auch Planeten außerhalb unseres Sonnensystems (extrasolare Planeten oder kurz Exoplaneten) gibt, aber man konnte solche nicht beobachten oder indirekt nachweisen. Erst 1992 gelang der erste bestätigte Nachweis eines Planeten um einen Pulsar. Zeitliche Änderungen der Puls-Periode ließen auf umlaufende Planeten schließen [WF92].

Diese Entdeckung machte die Suche nach weiteren extrasolaren Planeten zu einem beliebten Thema der Astronomie und viele weitere Entdeckungen folgten. Mittlerweile listet das NASA Exoplanet Archive mehr als 3700 bestätigte Entdeckungen auf [NAS18a]. Besonders die Suche nach erdähnlichen extrasolaren Planeten in der habitablen Zone des Sterns stellt hier ein interessantes Teilgebiet dar und kann mittlerweile einige Funde aufweisen.

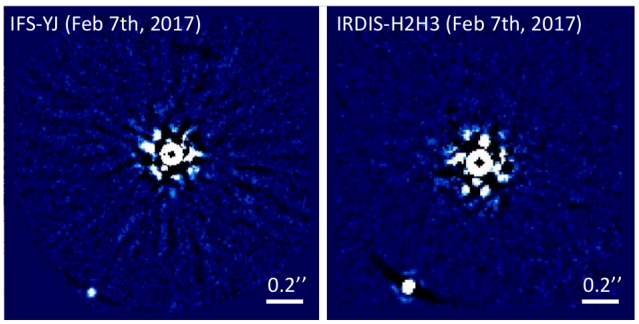

Abbildung 1.1.: Direct Imaging HIP 65426 b [CDL+17]

Der Planet links unten ist im Abstand von etwa 92 AU zum herausgefilterten Stern in der Mitte zu sehen.

Seit 2004 gelingen Entdeckungen auch durch direkte Beobachtungen mit bildgebenden Verfahren [CLD+04]. Ein aktueller Fund [CDL+17] ist in Abbildung 1.1 zu sehen. Hier sehen wir den Planeten bereits in mehrere Pixel aufgelöst.

Ein Großteil aller Entdeckungen wurde mithilfe des Weltraumteleskops Kepler getätigt, das aber mittlerweile am Ende seiner Missionsdauer angelangt ist [NAS18b]. Ein Nachfolger, das Weltraumteleskop TESS, ist im April 2018 gestartet und setzt die Suche nun großflächiger und mit modernerer Technik fort [NAS18c].

Für Mitte der 2020er Jahre ist der Start des Weltraumteleskops WFIRST angesetzt [NAS18d] und man beschäftigt sich bereits mit Methoden, um damit Atmosphären von extrasolaren Planeten zu analysieren [LSB18].

Die Suche nach extrasolaren Planeten wird also weiter verstärkt betrieben. Die Zahl der Entdeckungen, die Vielfalt unterschiedlichster Planeten und auch die Menge an Detailinformationen wachsen mit fortschreitender Technik.

Abbildung 1.2.: Weltraumteleskop WFRIST [NAS18d]

1.1. Motivation

Zur Charakterisierung extrasolarer Planeten ist es erforderlich, die beobachteten Erscheinungen wie Spektren, Helligkeiten, Dopplereffekte und zeitliche Schwankungen davon anhand von Modellrechnungen zu erklären. Entfernung, Masse, Radius, Zusammensetzung und weitere charakteristische Größen von Stern und Planet sind Parameter solcher Modelle. Es gilt dann, die Parameter dieses Modells so zu bestimmen, dass berechnete Größen mit minimiertem Fehler den beobachteten gleichen.

Abhängig von der zur Entdeckung eingesetzten Methode (Pulsar Timing, Radial Velocity, Transit, Direct Imaging, Gravitational Microlensing, etc.) lassen sich aus der Beobachtung direkt weitere Parameter wie Umlaufzeit oder Radius (aus der Verdeckung des Sterns) des Planeten folgern. Basiert das Verfahren auf der vom Planeten verursachten Bewegung des Sterns (und nicht auf dessen Verdeckung), liefert dies Hinweise auf die Masse des Planeten. Mit relativ einfachen Modellen lassen sich, ggf. unter Vernachlässigung von atmosphärischen Effekten, die Beobachtungsdaten aus den Modellparametern berechnen. [Wik18b]

Vor allem in bildgebenden Verfahren spielt aber die Atmosphäre eine Rolle, denn es wird das vom Planeten kommende Licht beobachtet, das ursprünglich vom Stern ausgestrahlt wurde. Der komplette Pfad der Strahlung ist hier relevant. Inzwischen möchte man daher in den Modellen auch die Atmosphäre der Planeten berücksichtigen, auch um diese sogar weiter zu analysieren. Dazu werden Strahlungstransport-Modelle benötigt, die darstellen wie das vom Stern kommende Licht auf den Planeten trifft und durch Atmosphäre und Oberfläche absorbiert oder gestreut wird, um dann schließlich das System in Richtung des Beobachters zu verlassen. Erschwerend kommt hinzu, dass dieses Verhalten je nach Wellenlänge des Lichts variiert. [Wik18c]

Ein Simulationsprogramm, mit dem solche Atmosphären-Modelle berechnet werden, ist PHOENIX [Ham18], das an der Hamburger Sternwarte entwickelt wird. PHOENIX ist in Fortran geschrieben und für den Einsatz im Hochleistungsrechnen konzipiert. Es wurde unter anderem auch im oben bereits genanntem [CDL+17] als Atmosphären-Modell verwendet.

Bei fortschreitender Beobachtungstechnik sind entsprechend komplexe Modelle notwendig, deren Berechnung auch beim Einsatz eines Hochleistungsrechners viel Zeit in Anspruch nimmt. In dieser Arbeit wird versucht, rechenaufwendige Programmteile von PHOENIX aufzudecken und die Leistung zu verbessern.

1.2. Geplantes Vorgehen und Einschränkungen

Zunächst einmal muss ein Verständnis der Konzepte und Abläufe von PHOENIX aufgebaut werden. Dann stellt allein das Kompilieren von PHOENIX mithilfe verschiedener Compiler und auf unterschiedlichen Systemen eine zu bewältigende Aufgabe dar, da Abhängigkeiten zu verschiedenen Bibliotheken und die bereits eingesetzten Parallelisierungs-Techniken (MPI, OpenMP, Vektorisierung) stets Ursache von Inkompatibilitäten darstellen können.

PHOENIX stellt eine Vielzahl von Methoden bereit. Im Rahmen dieser Arbeit wird nur ein einzelner Aspekt betrachtet werden, der zur Simulation extrasolarer Planeten besonders

relevant ist: der Strahlungstransport in 3D (3DRT), bei dem der Planet als sphärisches Gitter unter Verwendung von Kugelkoordinaten dargestellt wird. Näheres zum sphärischen Gitter wird im Abschnitt 2.3 beschrieben. Außerdem unterscheidet PHOENIX theoretisch zwischen langen und kurzen Charakteristiken, die Implementation konzentriert sich allerdings auf erstere. Daher ist mit dem Fokus auf einen konkreten Programmteil hier die Wahl ebenfalls auf lange Charakteristiken gefallen und kurze werden ist dieser Arbeit nicht weiter betrachtet. Näheres zu Charakteristiken wird in Kapitel 2 beschrieben.

Weiterhin bietet PHOENIX einen reduzierten Programmablauf zu Testzwecken an, bei dem von vielen grundsätzlich ähnlichen Berechnungen lediglich ein kleiner Teil durchgeführt wird. Details hierzu sind in Abschnitt 2.2 zu finden. Alle Programmläufe dieser Arbeit werden nur solche Tests sein. Es ist zu erwarten, dass eine Leistungssteigerung gemessen an einem solchen Test auch einen analogen Effekt in einem vollständigen Programmlauf erzielt.

Aus einer Reihe solcher Testläufe wird zunächst der Ist-Zustand ermittelt. Verschiedene Compiler werden miteinander anhand des erzielten Speedups in Abhängigkeit zur eingesetzten Hardware verglichen. Hier wird unterscheiden, ob die Rechenleistung in weitere Prozesse oder Threads investiert wird. Dies soll nur einen groben Einblick in das Skalierungsverhalten von PHOENIX liefern und eine Referenz für später folgende Änderungen darstellen.

Ziel dieser Arbeit soll nicht sein, auf einem konkreten System durch Variation der Komponenten (Compiler und Bibliotheken in jeweils verschiedenen Versionen und Ausprägungen) sowie verschiedenster Compiler-Optionen ein Optimum der Leistung zu erzielen. Solche Optimierungen sind natürlich sinnvoll und auch dringend anzuraten, sofern das Zielsystem feststeht und umfangreiche, produktive Programmläufe anstehen. Mit neuen Versionen der Komponenten verlieren solche Ergebnisse aber auch schnell ihre Aussage.

Es wird davon ausgegangen, dass der betrachtete Programmteil inhaltlich ausgereift, die eingesetzten Algorithmen aber noch nicht hinsichtlich der Leistung optimiert sind. Kern dieser Arbeit soll daher sein, Ansätze zu liefern, die unabhängig von eingesetzten Komponenten ein grundlegendes Potential zur Leistungssteigerung darstellen.

Mittels Profiling wird aufgezeigt, welche Funktionen besonders häufig aufgerufen werden oder besonders viel Laufzeit beanspruchen. Dies soll Hinweise auf die Engpässe im Programm liefern und Code-Abschnitte zur weiteren Analyse in den Fokus rücken.

Veränderungen am Quellcode werden anhand anschließender Laufzeitmessungen mit den Referenzkurven des Originals verglichen und bewertet. Ziel ist es natürlich, Varianten, die zu einer Leistungssteigerung führen, hervorzuheben und zu übernehmen.

2. PHOENIX

PHOENIX ist ein moderner Code zur Modellrechnung von Stern- und Planeten-Atmosphären. Hauptsächlich für Atmosphären von Sternen diverser Klassen entwickelt, kann er aber auch für Atmosphären von großen extrasolaren Planeten eingesetzt werden, da sich die Modelle gleichen. Entwickelt wird PHOENIX von der Arbeitsgruppe von P. Hauschildt an der Hamburger Sternwarte. [Ham18]

Für die Modellierung von dichten Atmosphären spielt der Strahlungstransport eine bedeutende Rolle und kann, wie in Ansätzen aus der Hydrodynamik, nicht vereinfacht dargestellt werden. Die Strahlungstransportgleichung, mit einem wiederum von der Strahlung abhängigen Term für die Streuung, ist eine Integro-Differentialgleichung und nur mit hohem Aufwand numerisch lösbar. [HB06, Wik18c]

Die zu modellierende Atmosphäre wird als räumlich diskretisiertes Gitter dargestellt, innerhalb jedes Gitterelements wird ein linearer Verlauf angenommen. Die Lösung der Strahlungstransportgleichung erfolgt dann auf Charakteristiken (Geraden, die durch die Gitterelemente geführt werden), mit der 'Operator Splitting' (OS) Methode und wird in [HB06, See08, Squ16] beschrieben. Hier sei nur erwähnt, dass es sich um ein Iterationsverfahren handelt, in dem ein Operator Λ^* auf jedes Gitterelement wirkt und sich der nächste Iterationsschritt aus dem aktuellen und dem der nächsten Nachbarelemente berechnet.

Aus den Ergebnissen können dann synthetische Spektren generiert und mit Beobachtungen verglichen werden. Daraus lassen sich dann Erkenntnisse, z.B. über die Zusammensetzung der Atmosphären, gewinnen.

2.1. Vollständige Simulationen

In der zuvor angedeuteten Lösung der Strahlungstransportgleichung wird zunächst von einer vorgegebenen Wellenlänge λ der Strahlung ausgegangen, da die Absorption von Strahlung, die in die Berechnungen eingeht, abhängig von der Wellenlänge ist. Um ein synthetisches Spektrum zu erhalten, muss also der betrachtete Wellenlängenbereich in eine Reihe diskreter Wellenlängen aufgelöst werden. Jede einzelne Wellenlänge λ kann dann i.d.R. unabhängig von den anderen betrachtet und im Modell berechnet werden. Hier erfolgt daher die erste Ebene der Parallelisierung: Teile der zur Verfügung stehenden Rechenknoten können unabhängig voneinander jeweils eine Teilmenge der Wellenlängen betrachten. Mit zunehmender Auflösung des Gitters, auf dem das Problem gelöst wird, steigt der Speicherbedarf (RAM) stark an, so dass ein Modell gar nicht im Hauptspeicher eines einzelnen Rechenknotens gehalten werden kann. Es ist also auch daher schon notwendig, mehrere Knoten zusammenzufassen, die gemeinsam das Problem für eine einzelne Wellenlänge lösen. Diese zusammengefassten Knoten müssen

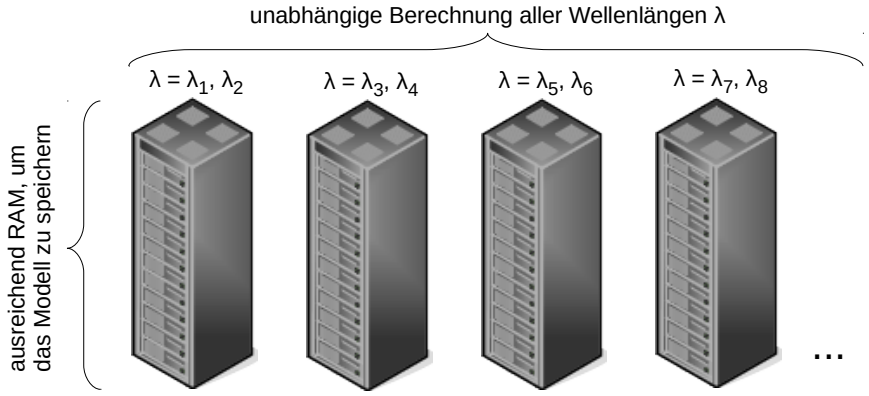

Abbildung 2.1.: MPI Parallelisierung Ebene 1
Jede Modellrechnung wird über mehrere Knoten verteilt, da in der Regel der RAM eines Knoten nicht ausreicht. Unabhängig voneinander werden die Modelle zu allen erforderlichen diskretisierten Wellenlängen λ, über die zur Verfügung stehende Hardware verteilt, berechnet.

zur Berechnung miteinander kommunizieren, wozu das Message Passing Interface (MPI)[1] als Methode für verteilten Speicher verwendet wird. Diese erste Ebene der Parallelisierung ist in Abbildung 2.1 dargestellt. [See08, HB06]

Wie zuvor beschrieben erfolgt die Berechnung entlang von Charakteristiken, die die Gitterelemente in einem vorgegebenen Raumwinkel durchlaufen. Idealerweise soll das Modell aus möglichst jeder Richtung betrachtet werden. Auch hier ist also wieder eine diskrete Auswahl der Raumwinkel (θ, ϕ) zu treffen. In dieser Arbeit wurde hierzu stets ein Gitter von 16×16 Werten verwendet. Die zweite Ebene der Parallelisierung teilt also die Wertepaare (θ, ϕ) auf die zur Verfügung stehenden Prozesse auf, wie in Abbildung 2.2 angedeutet. Ebenso wie auf Knoten-Ebene, wird auch auf Prozess-Ebene MPI als Parallelisierungstechnik verwendet. [Squ16]

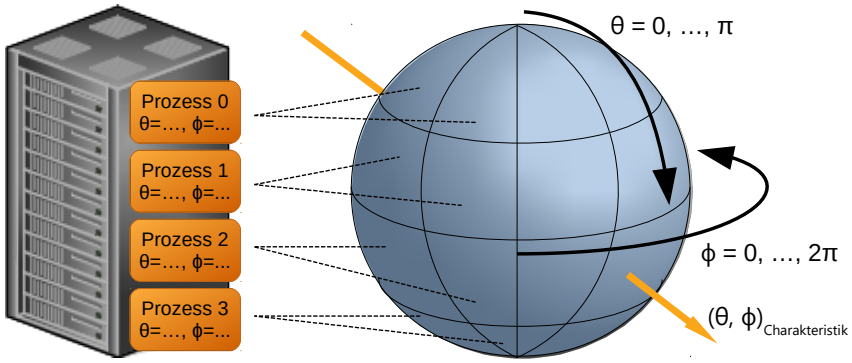

Abbildung 2.2.: MPI Parallelisierung Ebene 2
Charakteristiken beschreiben die Strahlungsausbreitung in Richtung eines Raumwinkels (θ, ϕ). Alle zu betrachtenden Raumwinkel werden auf die vorhandenen Prozesse aufgeteilt.

[1] MPI Standard: `https://www.mpi-forum.org`

Als Teilaufgabe des Λ^* Solvers eines einzelnen Prozesses werden sogenannte Tracker ausgeführt. Der Tracker hat zunächst die Aufgabe, Charakteristiken im vorgegebenen Raumwinkel zu erstellen, so dass jedes Element des räumlichen Gitters mindestens von einer Charakteristik durchlaufen wird. Die namensgebende Aufgabe eines Trackers ist dann das Verfolgen der Charakteristiken von einem Gitterelement zum nächsten, so dass in jedem Gitterelement auch ein Rechenschritt erfolgt. Um die Strahlungstransportgleichung zu lösen, führt der Tracker diese Berechnungen für die einzelnen Gitterelemente bzw. Segmente einer Charakteristik aus, die zu dieser Iteration und diesem Raumwinkel gehören und bereitet so die weiteren Schritte des Λ^* Solvers vor.

In PHOENIX gibt es eine Vielzahl verschiedener Tracker, die sich in ihrer Methodik unterscheiden. Ein Unterscheidungsmerkmal ist die Art, das Problem durch ein räumliches Gitter zu beschreiben. Hier gibt es Gitter in kartesischen Koordinaten, aber auch Zylinder- und Kugelkoordinaten. Ein weiteres Merkmal beschreibt die Charakteristiken. Lange Charakteristiken (LC) durchlaufen das Gitter als durchgehende Gerade, kurze Charakteristiken (SC) werden von der Mitte eines Gitterelements in das nächste geführt, dort setzt dann aber ein neues Teilstück wieder in dessen Mitte an, dies ist in [HB06, Squ16] anschaulich dargestellt. Schließlich können sich ansonsten gleichartige Tracker auch einfach in der mathematischen Methode oder den verwendeten Algorithmen unterscheiden.

In der Auswahl an Trackern sind einige auch bereits in einer dritten Ebene weiter parallelisiert. Hier werden dann Parellelisierungsmethoden auf gemeinsamem Speicher verwendet, wie Multithreading, was besonders komfortabel mittels OpenMP[1] implementiert werden kann. Auch von den Prozessoren bereitgestellte Vektor-Befehlssätze (AVX)[2] [3] werden verwendet, sofern PHOENIX mit einem dazu fähigen Compiler und für eine entsprechende Zielarchitektur erstellt wird.

Im Folgenden wird ein LC Tracker für ein sphärisches Gitter unter Verwendung von Kugelkoordinaten betrachtet.

2.2. Reduzierte Testläufe

Um die Funktion eines Trackers zu Testen oder die Leistung zu bewerten, ist es ausreichend, sich auf lediglich eine Wellenlänge zu beschränken, da die Berechnungen zu unterschiedlichen Wellenlängen unabhängig voneinander sind. Hierzu bietet PHOENIX einen separaten Test-Programmeinstieg über `program main` in `Source/3DRT/main.f` an. Die erste Ebene der Parallelisierung wird dabei übergangen. In dieser Arbeit werden nur solche Test-Programmläufe verwendet.

Für den Aufruf des ausgewählten sphärischen Trackers sieht der stark vereinfachte Ablauf dann aus wie in Abbildung 2.3 schematisch dargestellt. Wichtig zu bemerken ist, dass der Tracker aus dem Solver aufgerufen wird und dieser wiederum innerhalb einer Schleife über die Raumwinkel (θ, ϕ). Eine weitere äußere Ebene stellt eine Schleife über die Iterationen dar, bis die Maximalzahl oder die gewünschte Genauigkeit erreicht ist (letzteres ist im Diagramm

[1] OpenMP Spezifikation: `https://www.openmp.org`
[2] Advanced Vector Extensions: `https://de.wikipedia.org/wiki/Advanced_Vector_Extensions`
[3] AVX-512 Instructions: `https://software.intel.com/en-us/blogs/2013/avx-512-instructions`

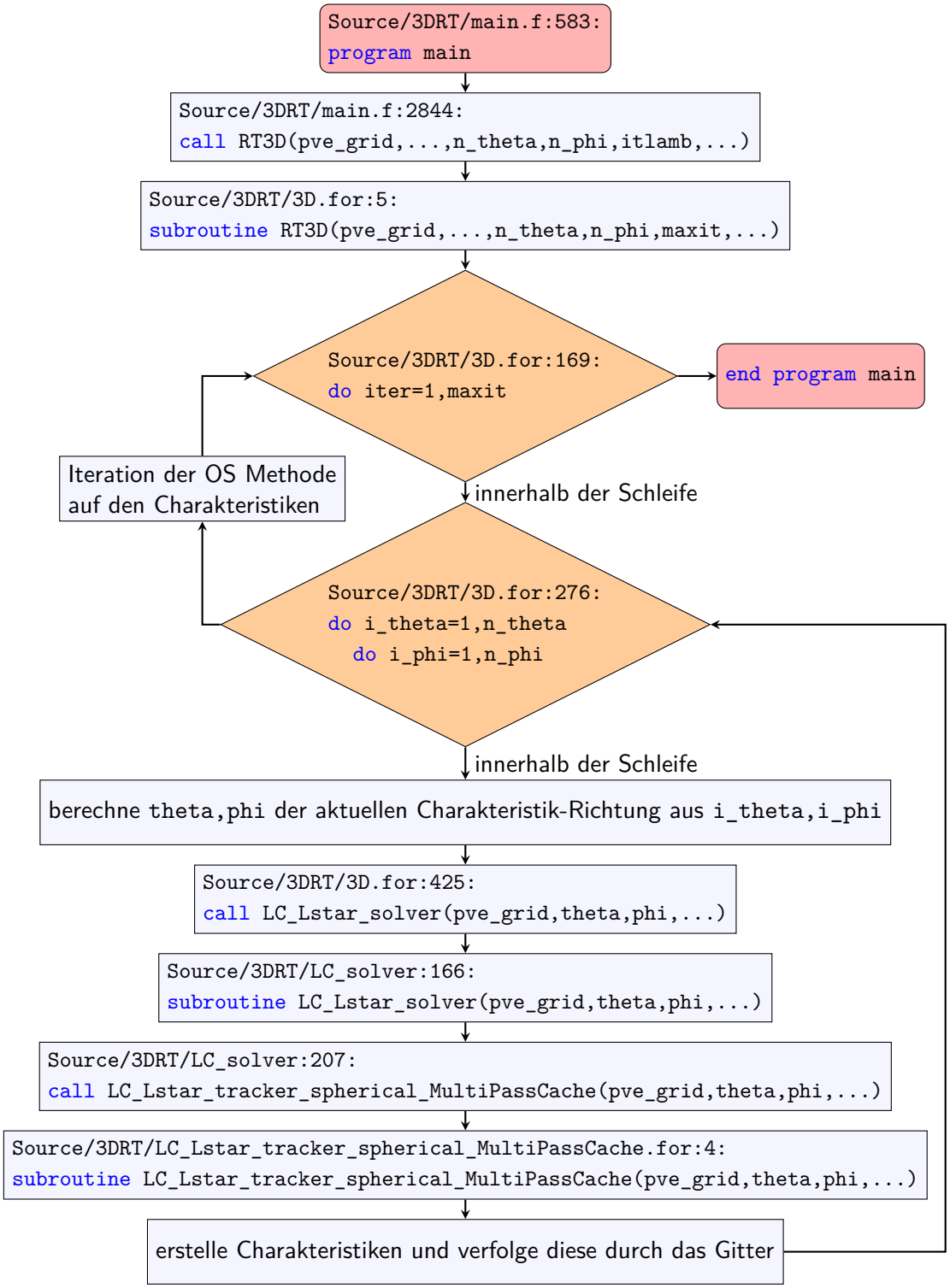

Abbildung 2.3.: Schematische Darstellung der Aufrufe bis hin zum sphärischen Tracker
Dies ist der verkürzte Ablauf für einen Test. Für die Zahl der Iterationen existiert neben der maximalen Anzahl `maxit` auch noch eine Abbruchbedingung nach Genauigkeit. Die angegebenen Zeilennummern entsprechen dem Git Commit 5bcab32 vom 21.03.2018.

nicht explizit dargestellt, in PHOENIX aber als zusätzliche Abbruchbedingung der Schleife implementiert).

2.3. Sphärischer Tracker

Der Tracker `LC_Lstar_tracker_spherical_MultiPassCache()` wird für LC in der Λ^* Methode verwendet, wenn das bestrahlte Objekt durch ein räumliches Gitter auf Kugelkoordinaten dargestellt wird. Dies ist eine für Sterne oder extrasolare Planeten geeignete Darstellung. Zunächst wird das Objekt in mehrere Schichten unterschiedlicher Radien eingeteilt, indiziert durch i_r. Diese beginnen bei einem innersten Radius, der in der Regel von Null verschieden sein wird, schließlich soll die Atmosphäre modelliert werden und nicht der dichte Kern, durch den keine Strahlung dringt. Die konkreten Radien der Schichten müssen hierbei auch nicht äquidistant verteilt sein, sondern der Index i_r verweist auf einen frei wählbaren Eintrag in einem Array `Rmap`, so dass sie z.B. nach innen dünner angelegt werden können. Die Schichten werden weiterhin gemäß der Kugelkoordinaten in Segmente zerlegt, indiziert durch i_θ und i_ϕ.

Um Verwechslungen zwischen den Raumwinkeln (θ, ϕ) der Charakteristiken und der Einteilung des Gitters zu vermeiden, wird in dieser Arbeit über das Gitter stets in der indizierten Darstellung (i_r, i_θ, i_ϕ) gesprochen.

Die Anzahl der Gitterelemente wird über (n_r, n_θ, n_ϕ) vorgegeben, wobei zu beachten ist, dass der Index i_r von $-n_r$ bis n_r läuft, die anderen Indizes entsprechend. So liegt also der Index 0 jeweils in der Mitte des Intervalls und ein Gitter mit $n_r = n_\theta = n_\phi = n$ hat $(2n+1)^3$ Elemente.

Abbildung 2.4 zeigt einen Schnitt durch das Gitter, die θ-Komponente ist nicht dargestellt. Es ist offensichtlich, dass die einzelnen Gitterelemente sehr abweichende Ausmaße annehmen.

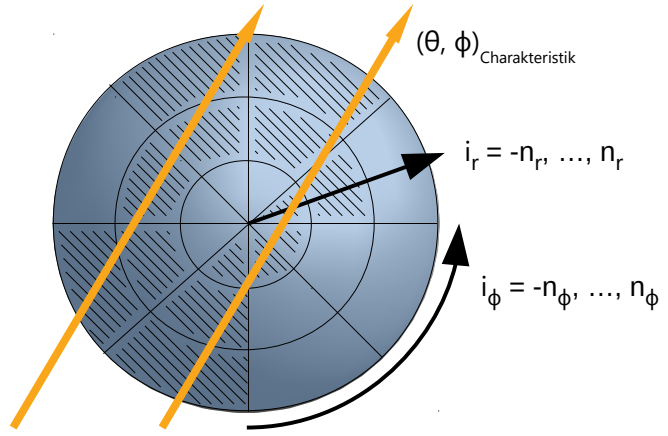

Abbildung 2.4.: Charakteristiken durchlaufen das sphärische Gitter
Angedeutet sind mehrere Charakteristiken, die das sphärische Gitter im vorgegebenen Raumwinkel (θ, ϕ) durchlaufen. Gitterelemente, indiziert durch (i_r, i_θ, i_ϕ), werden von diesen getroffen und deren Inhalt berechnet. Hier gilt es gerade ausreichend Charakteristiken zu erzeugen, so dass alle Gitterelemente getroffen werden. Nicht abgebildet ist die dritte Dimension $i_\theta = -n_\theta, ..., n_\theta$.

Als Resultat ist die Aufgabe, die Charakteristik durch das Gitter zu führen, anspruchsvoller als in einem kartesischen Gitter und wird durch eine adaptive Schrittweite angegangen. Ebenso ist es in dem sphärischen Gitter schwieriger, eine geeignete Menge an Charakteristiken zu wählen, so dass alle Gitterelemente einmal durchlaufen werden, da die Charakteristiken ja nach wie vor Geraden im kartesischen Koordinatensystem sind. [HB09]

Das `MultiPass` im Namen des Trackers besagt, dass in mehreren, alle Gitterelemente umfassenden, Schleifen jeweils einzelne Teilaufgaben ausgeführt werden. Der im Namen erwähnte `Cache` ist optional und kann über `use_tracking_cache = f` im Jobscript auch abgeschaltet werden, wohingegen die Angabe von `use_MultiPass = f` zur Verwendung eines anderen Trackers führen würde. Der Sinn des Cache ist es, die wie zuvor erwähnt schwierige Aufgabe, eine geeignete Menge an Charakteristiken zu platzieren, nicht in jeder Iteration erneut durchzuführen sondern im RAM zwischenzuspeichern. Für andere Raumwinkel (θ, ϕ) unterscheidet sich die Geometrie allerdings und es müssen mehr oder weniger Charakteristiken an anderen Startpunkten platziert werden. Das heißt, dass der Cache alle Informationen enthalten muss, die notwendig sind, um eine Charakteristik zu reproduzieren, multipliziert mit der Gesamtzahl der Charakteristiken über alle Raumwinkel. Je größer das Gitter, desto größer auch die Anzahl der benötigten Charakteristiken, so dass zur Verwendung des Cache der Bedarf an RAM stark steigt. Ist aber genügend RAM vorhanden, so sollte das Vermeiden eines sich wiederholenden komplexen Arbeitsschritts eine deutliche Ersparnis der Laufzeit bedeuten.

Diese Variante des Trackers mit MultiPass und Cache ist Gegenstand der aktuellen Entwicklung und verspricht eine Leistungssteigerung gegenüber einer herkömmlichen Variante. In dieser Arbeit wird dieser neuere Ansatz weiter nach zusätzlichem Potential untersucht.

2.4. Verwendung

Um PHOENIX zu kompilieren, kann es erforderlich sein, zunächst einige Anpassungen der Konfiguration vorzunehmen. Im Anhang A werden die Anpassungen beispielhaft für das Cluster des Arbeitsbereichs Wissenschaftliches Rechnen[1] gezeigt, auf dem die Schnittstellen der Mathe-Bibliothek LAPACK[2] durch die Bibliothek OPENBLAS[3] bereitgestellt werden. Außerdem zeigt das Beispiel die notwendige Konfiguration zur Verwendung des flang-Compilers, auf den später noch eingegangen wird.

Zur Ausführung von PHOENIX stehen Jobscripts bereit, die ebenfalls an die eigenen Anforderungen angepasst werden können. Die zu verwendenden Parameter werden mittels Fortran Namelist[4] vom Jobscript an PHOENIX übergeben. Einige für diese Arbeit wichtige Parameter werden in Tabelle 2.1 beschrieben. Die Jobscripts sind für den, auf Hochleistungsrechnern

[1] Arbeitsbereich Wissenschaftliches Rechnen: `https://wr.informatik.uni-hamburg.de`
[2] LAPACK: `http://www.netlib.org/lapack`
[3] OPENBLAS: `https://www.openblas.net`
[4] Fortran Namelist: `https://docs.oracle.com/cd/E19957-01/805-4939/6j4m0vnc6`

weit verbreiteten, SLURM[1] Workload Manager geeignet, aber auf Einzelsystemen auch als Shell-Script ausführbar.

Tabelle 2.1.: Parameterübergabe an PHOENIX

Parameter	Wert	Beschreibung
job	12	12: 3DRT mit LC Solver
use_spherical_grid	t	true für Start des ausgewählten Trackers
use_MultiPass	t	true für Start des ausgewählten Trackers
use_tracking_cache	t, f	Optionaler Cache der Charakteristik-Geometrie
fasttrack	f	false für Start des ausgewählten Trackers
use_2pass	f	false für Start des ausgewählten Trackers
n_phi	16	Anzahl der Raumwinkel in ϕ-Richtung
n_theta	16	Anzahl der Raumwinkel in θ-Richtung
nx	128, 8	Anzahl Gitterelemente n_r (beachte $2n+1$)
ny	16, 8	Anzahl Gitterelemente n_θ (beachte $2n+1$)
nz	32	Anzahl Gitterelemente n_ϕ (beachte $2n+1$)
OS_Solver	4	4: Jordan
itlamb	30	Maximale Iterationen des Solvers
epsacc	1d-6	Bedingung zum Abbruch der Iterationen nach Genauigkeit

Dies ist nur eine Auswahl aller möglichen Parameter. Die angegebenen Werte werden in dieser Arbeit verwendet sofern nicht anders angegeben.

[1] SLURM: `https://slurm.schedmd.com`

3. Grundlagen der Leistungsanalyse

Im Folgenden sollen die theoretischen Grundlagen zur Leistungsanalyse betrachtet werden, um im Anschluss konkrete Messungen bewerten zu können.

3.1. Speedup

Um das Leistungsverhalten eines Algorithmus oder einer konkreten Implementation allgemeingültig zu betrachten, wählt man möglichst eine Darstellung, die unabhängig von der eingesetzten Hardware ist. Man verwendet daher statt der gemessenen Programm-Laufzeit $T(p)$ bei Verwendung von p Prozessen (oder Threads) den Speedup $S(p)$, der die Laufzeit in Relation zur Laufzeit T^* des optimalen sequentiellen Programms setzt: [RR12]

$$S(p) = \frac{T^*}{T(p)} \tag{3.1}$$

Selbstverständlich wird in der Definition davon ausgegangen, dass stets dasselbe Problem, insbesondere dieselbe Problemgröße betrachtet wird. Es wird außerdem streng unterschieden zwischen der Laufzeit des originalen sequentiellen Programms T^* und der eines angepassten parallelisierten Programms, das lediglich mit einem Prozess ausgeführt wird, $T(1)$. Weiterhin wird immer vom optimalen sequentiellen Programm ausgegangen. In den Speedup sollen also nicht während der Anpassungen vorgenommene Verbesserungen am Algorithmus eingehen, die auch im sequentiellen Fall umgesetzt werden könnten (aber nicht umgesetzt werden). Wenn, so wie in dieser Arbeit, ein sequentielles Programm gar nicht vorliegt, so handelt es sich bei Verwendung von $T^* = T(1)$ um eine Näherung. Das gleiche gilt für den häufig eintretenden Fall, dass ein sequentielles Programm vorliegt, bei dem aber nicht bekannt ist, ob es sich um die optimale Lösung handelt. [RR12]

Der ideale Fall, dass sich die Arbeit eines Programms ohne Mehraufwand auf p Prozesse aufteilen lässt, soll durch den optimalen Speedup $S_{optimal}(p)$ dargestellt werden. Jeder Prozess hat in diesem Fall nur den Anteil $\frac{1}{p}$ der gesamten Arbeit zu bewältigen und wird dies in der Laufzeit $\frac{T^*}{p}$ schaffen. Da nun alle Prozesse zeitgleich ablaufen, wird also auch das gesamte Programm eine Laufzeit von $\frac{T^*}{p}$ haben. Somit gilt im optimalen Fall für den Speedup:

$$S_{optimal}(p) = \frac{T^*}{T(p)} = \frac{T^*}{(\frac{T^*}{p})} = p \tag{3.2}$$

Dieser optimale Speedup wird in der Regel aber nie erreicht werden, da stets zusätzlicher Aufwand für das Aufteilen der Arbeit anfällt. Weiterhin ist davon auszugehen, dass immer nur ein Bruchteil parallelisierbar ist und stets ein sequentieller Anteil f verbleibt. Dann setzt

sich auch die Laufzeit aus einem sequentiellen und einem parallelen Anteil zusammen, so dass für den Speedup das Amdahlsche Gesetzt gilt: [RR12]

$$S_f(p) = \frac{T^*}{fT^* + \frac{1-f}{p}T^*} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f} \tag{3.3}$$

Der Speedup realer Anwendungen wird also nicht dem optimalen linearen Fall entsprechen, sondern wird sich mit zunehmender Zahl der Prozesse p weiter davon entfernen. Am Verlauf der Kurve $S(p)$ lässt sich dennoch erkennen, wie gut die eingesetzte Hardware, im Vergleich zum theoretisch möglichen, ausgelastet wird.

3.1.1. Berechnung des Messfehlers

Um aus Messungen robuste Erkenntnisse zu gewinnen, sollten die Auswertungen stets auf mehreren Messungen zu einem identischen Satz von Parametern basieren. Zur Bestimmung des Speedup bedeutet das, dass zu einer gegebenen Anzahl Prozesse p mehrere Programmläufe durchgeführt werden und aus den erhaltenen N Messwerten der Laufzeit T_i für $i = 1, ..., N$ dann der Mittelwert $\bar{T}$ und die Standardabweichung σ_T bestimmt werden: [Wik18a]

$$\bar{T} = \frac{1}{N}\sum_{i=1}^{N} T_i \tag{3.4}$$

$$\sigma_T = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(T_i - \bar{T})^2} \tag{3.5}$$

Die daraus abgeleitete Größe Speedup ist dann mit einer entsprechenden Abweichung behaftet, die sich nach den Regeln der Fehlerfortpflanzung berechnen lässt. [FP10]

$$\sigma_{S(p)} = \left|\frac{\partial S(p)}{\partial T^*}\right| \sigma_{T^*} + \left|\frac{\partial S(p)}{\partial T}\right| \sigma_T \tag{3.6}$$

Aus (3.1) folgt

$$\bar{S}(p) = \frac{\bar{T}^*}{\bar{T}(p)} \tag{3.7}$$

$$\frac{\partial S(p)}{\partial T^*} = \frac{1}{T(p)} \quad , \quad \frac{\partial S(p)}{\partial T} = -\frac{T^*}{(T(p))^2} \tag{3.8}$$

Und damit folgt aus (3.6)

$$\sigma_{S(p)} = \frac{1}{T(p)}\sigma_{T^*} + \frac{T^*}{(T(p))^2}\sigma_T \tag{3.9}$$

3.1.2. Grafische Darstellung

Der ermittelte Speedup wird üblicherweise in einem Liniendiagramm dargestellt, indem auf der X-Achse die Zahl der Prozesse, Threads oder das Produkt aus beiden (hier als Recheneinheiten bezeichnet) und auf der Y-Achse der Speedup aufgetragen wird. Die einzelnen Messwerte sollten als solche zu erkennen sein, indem sie als Punkte dargestellt werden. Der theoretisch mögliche lineare Speedup kann zur Orientierung ebenfalls als Hilfslinie dargestellt werden. [BM06, HW10]

Als Messwerte gelten die Mittelwerte mehrerer Messungen. Die draus errechnete Standardabweichung des Speedups wird über Fehlerindikatoren dargestellt, um auch im Diagramm mögliche Schwankungen der Messwerte zu zeigen.

3.1.3. Auswertung

Eine Anwendung skaliert gut, wenn ihr Speedup dem optimalen nahe kommt oder eine zwar reduzierte aber dennoch konstante Steigung aufweist. Durch ein Speedup-Diagramm sollen Bereiche, in denen der Speedup fällt oder in eine Sättigung übergeht, zur weiteren Betrachtung aufgezeigt werden.

Ist die dargestellte Abweichung sehr hoch (etwa zweistellige Prozentwerte vom Mittelwert), kann die Messung schon mal auf technische Probleme untersucht und ggf. wiederholt werden. Schwankungen sind aber durchaus zu erwarten und sollen als solche in den Ergebnissen dargestellt werden. Der Betrachter eines Speedup-Diagramms kann als realen Verlauf eine Interpretation vornehmen, die die gezeigten Messpunkte innerhalb der Fehlerindikatoren verfehlt.

3.1.4. Strong und Weak Scaling

Die zuvor beschriebene Skalierung bei Betrachtung einer festen Problemgröße wird Strong Scaling genannt und beschreibt die Möglichkeiten mit mehr Hardwareeinsatz immer kürzere Laufzeiten zu erreichen. Da die Laufzeit nicht beliebig zu verkleinern ist, kann man stattdessen auch ein sogenanntes Weak Scaling betrachten. Hierbei wird die Problemgröße zusammen mit dem Hardwareeinsatz vergrößert, so dass man bei $M \cdot p$ Prozessen ein um Faktor M größeres Problem in gleicher Zeit lösen können sollte. [HW10]

Die Vergrößerung des Problems um Faktor M heißt aber nicht, dass es einen problembeschreibenden Parameter N gibt, für den nun $M \cdot N$ eingesetzt wird. Hier gilt es die Komplexität des Problems zu betrachten. Handelt es sich z.B. um ein $O(N^2)$ Problem, wäre $\sqrt{M} \cdot N$ anzusetzen.

3.2. Profiling und Tracing

Neben der Gesamtlaufzeit eines Programms ist zur Optimierung besonders interessant, welche Teilabschnitte des Programms den größten Anteil daran haben. “Eine alte Faustregel besagt, dass ein Programm ca. 90% seiner Laufzeit in 10% des Programmcodes verbringt” [BM06]. Hieraus lässt sich Ableiten, auf welche Abschnitte bei der Optimierung dann besonderes

Augenmerk gelegt werden sollte. Um an diese Informationen zu gelangen, werden Profiler eingesetzt. Bekannte Exemplare sind gprof[1] und Score-P[2].

Es existieren zwei Methoden des Profilings. Beim Sampling wird das laufende Programm regelmäßig unterbrochen und notiert, an welcher Stelle es sich gerade befindet. Dies ist eine statistische Methode, die bei genügend langer Laufzeit brauchbare Aussagen bis hinunter zu einzelnen Code-Zeilen erzeugen kann. Die Methode der Instrumentierung setzt am Compiler an, der Anweisungen einfügt, die das Betreten und Verlassen von Funktionen protokollieren. Hierbei wird tatsächlich jeder Aufruf erfasst, so dass hier auch schon vergleichsweise kurze Programmläufe Informationen liefern, die aber stets Funktionen als ganze Einheit betrachten. [HW10, BM06]

Werden bei der Instrumentierung nur Zähler für jeden Aufruf hochgezählt und Aufenthaltszeiten in der jeweiligen Funktion addiert, genügt dies für ein Profiling. Behält man hingegen jedes einzelne Ereignis und versieht es mit einem Zeitstempel, so spricht man vom Tracing. Geeignete Programme wie Vampir[3] können dann den zeitlichen Ablauf der aufgerufenen Funktionen für jeden Prozess visualisieren. Stellt man die Abläufe vieler beteiligter Prozesse gemeinsam dar, lässt sich aus dem Tracing das Zusammenspiel beurteilen, wie dies beispielhaft in Abbildung 3.1 zu sehen ist. Dies ist insbesondere hilfreich um ggf. aufzudecken, dass Prozesse lange Zeit nicht produktiv arbeiten können, weil sie auf Nachrichten von anderen warten müssen. Solche Probleme treten besonders dann auf, wenn eine Synchronisation stattfindet oder komplexe Kommunikations-Schemata der Prozesse untereinander am Werk sind. [Lud18]

Abbildung 3.1.: Beispiel eines Tracings, visualisiert in Vampir
Hier handelt es sich nicht um PHOENIX sondern um ein Beispiel. Grün dargestellt sind produktive, rechenintensive Phasen der Prozesse, rot dagegen sind MPI Kommunikation, vor allem die damit verbundenen Wartezeiten. Linien zeigen, welche Prozesse untereinander kommunizieren. Solch eine Darstellung kann Probleme des Kommunikationsschemas aufzeigen.

[1] gprof: `http://sourceware.org/binutils/docs/gprof`
[2] Score-P: `http://www.vi-hps.org/projects/score-p`
[3] Vampir: `https://vampir.eu/`

4. Leistungsanalyse Ist-Zustand

In diesem Kapitel soll der Ausgangszustand in PHOENIX bezüglich seiner Leistung analysiert werden. Ziel ist es, Ansatzpunkte zu möglichen Verbesserungen und eine Referenz zur erneuten Leistungsbewertung nach jeglichen Änderungen zu liefern. Als Ist-Zustand wird der Git Commit 5bcab32 vom 21.03.2018 der Entwicklungsversion `phxexp` betrachtet. Ausgeführt wird PHOENIX exemplarisch auf folgenden Systemen:

West

- Cluster des Arbeitsbereichs Wissenschaftliches Rechnen[1]
- 10 Knoten mit jeweils:
- 2 $\times$ Intel(R) Xeon(R) CPU X5650 @ 2.67GHz
- 12 Kerne, Hyper-Threading mit 2 Threads pro Kern
- 12 GB RAM
- 2 $\times$ 1 Gbit Ethernet

Hummel

- Cluster des Regionalen Rechenzentrums[2]
- Wird häufig von der Arbeitsgruppe Hauschildt der Hamburger Sternwarte für PHOENIX Modellrechungen verwendet
- 316 Standard-Knoten mit jeweils:
- 2 $\times$ Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
- 16 Kerne, Hyper-Threading mit 2 Threads pro Kern
- 64 GB RAM
- QDR-Infiniband

4.1. Skalierung anhand des Speedups

Der erreichte Speedup soll als Kriterium für die Leistungssteigerung durch Parallelisierung dienen. Dieser wird für zunehmenden Einsatz an CPU-Ressourcen in Form von Prozessen oder Threads evaluiert und die mit dem Einsatz skalierende Leistungssteigerung betrachtet. Für den Einsatz von PHOENIX werden hier zwei Varianten des zu verwendenden sphärischen

[1] WR Cluster: `https://wr.informatik.uni-hamburg.de/teaching/ressourcen/start`
[2] Hummel Cluster: `https://www.rrz.uni-hamburg.de/services/hpc/hummel-2015.html`

Gitters festgelegt. Ein kleines Gitter mit $17 \times 17 \times 65$ Elementen $(n_r = 8, n_\theta = 8, n_\phi = 32)$ soll für kurze Testläufe verwendet werden. In dieser Größe ist es möglich, die als Referenz benötigte Laufzeit $T(1)$ bei einem Prozess und einem Thread zu ermitteln.

Ein großes Gitter mit $257 \times 33 \times 65$ Elementen $(n_r = 128, n_\theta = 16, n_\phi = 32)$ entspricht eher realistischeren Modellrechnungen. Hierbei überschreiten jedoch die Laufzeiten bei einem einzelnen Prozess und Thread das Limit des West Clusters, so dass die Referenz-Laufzeit $T(1)$ nur geschätzt werden kann. Für die Schätzung wird außerhalb des Messbereichs der optimale lineare Speedup angenommen, also eine Referenz-Laufzeit $T(1) = p_{min} \cdot T(p_{min})$ für die kleinste messbare Prozesszahl p_{min} (vgl. Abschnitt 3.1). Weiterhin stößt diese Variante für höhere Prozesszahlen mit dem Hauptspeicherbedarf an die Grenzen der vorhandenen Hardware, so dass auch an diesem Ende zum Teil Messdaten fehlen.

In jedem Fall wird der Referenzwert durch $T^* = T(1)$ angenähert, da kein sequentielles Programm vorliegt (vgl. Abschnitt 3.1). Um mit unterschiedlichen Compilern erzeugte Programm-Varianten untereinander vergleichen zu können, wird im Folgenden der Speedup 1 immer auf die jeweils schnellste Variante bei einer Recheneinheit bezogen. Die Originalversion gilt als Referenz bei späteren Vergleichen mit angepassten Programmvarianten.

4.1.1. Verschiedene Compiler

Auch wenn es nicht Ziel dieser Arbeit sein soll, ein Optimum unter Einsatz des bestmöglichen Compilers und seiner Optionen zu erreichen, so werden dennoch mehrere Compiler betrachtet. Dies soll vor allem ausschließen, dass beobachtete Effekte als allgemeingültig angenommen werden, aber eigentlich nur bei Verwendung des konkreten Compilers auftreten oder gar Fehler in gerade der verwendeten Compiler-Version sind. Ein ähnliches Ziel soll auch durch den Test auf mehreren Systemen erreicht werden.

Im Folgenden werden hauptsächlich der Intel und der GFortran Compiler verwendet, um PHOENIX zu erstellen, da beide bereits für die Entwicklung von PHOENIX erprobt sind. Aber auch flang, das Fortran Frontend des LLVM Frameworks, wurde evaluiert. Näheres zu flang wird kurz im Abschnitt 4.1.2 erläutert. Im Detail kommen folgende Versionen zum Einsatz.

Intel: Auf dem Hummel Cluster stellt der Intel Compiler[1] in der Version 2017.5 den derzeitigen Standard für PHOENIX dar. Auf dem West Cluster wurde das Intel Parallel Studio[2] in der entsprechenden Cluster Version 2017.5 mit Spack[3] installiert.

GFortran: Der GFortran[4] Compiler des GCC Pakets in der Version 7.3.0 steht auf dem West Cluster zur Verfügung und wurde daher in dieser Version gewählt.

flang: Das flang[5] Frontend des LLVM[6] Frameworks konnte erst in der laufenden Entwicklungsversion, Commit 45d7aeb vom 07.07.2018, zum Einsatz gebracht werden. Diese basiert auf Version 6.0 von LLVM.

[1] Intel Fortran Compiler: `https://software.intel.com/en-us/fortran-compilers`
[2] Intel Parallel Studio: `https://software.intel.com/en-us/parallel-studio-xe`
[3] Spack Package Manager: `https://spack.io/`
[4] GNU Fortran: `https://gcc.gnu.org/wiki/GFortran`
[5] flang: `https://github.com/flang-compiler/flang`
[6] LLVM: `https://llvm.org/`

In Abbildung 4.1 wird der Speedup durch Erhöhung der MPI Prozesse dargestellt, wenn PHOENIX mit dem jeweiligen Compiler erstellt wurde. Im direkten Vergleich erzeugt der Intel Compiler die PHOENIX Variante mit der kürzesten Laufzeit. Da hier aber die vorhandenen, als funktionstüchtig bekannten, PHOENIX-Architekturen verwendet werden, und diese sich z.B. in der Angabe des Optimierungs-Levels[1] unterscheiden (für Intel wird `-O2` , verwendet, für GFortran dagegen `-O` , was `-O1` entspricht), sind dies keine Aussagen zur allgemeinen Bewertung der Compiler.

Für den flang Compiler wurden keine Anpassungen der Compiler-Flags vorgenommen, er basiert auf den gleichen Einstellungen wie die GFortran Variante. Dennoch ist die mit flang erzeugte Variante schneller. Hier ist sicherlich noch Potential für weitere Leistungsgewinne durch Optimierung der Einstellungen.

Bei Verwendung des großen Gitters ist aber zu sehen, dass sowohl Intel als auch flang ein PHOENIX erzeugen, dass mehr Hauptspeicher erfordert. Denn dies ist der limitierende Faktor, der verhindert, dass diese hier mit mehr als 9 Prozessen eingesetzt werden konnten.[2] GFortran geht hier effizienter mit dem RAM um, so dass erfolgreich bis zu 12 Prozessen skaliert werden konnte. Auf mehr als 12 Prozesse zu erhöhen, war dann aber auch mit GFortran nicht möglich.

Die von den verschiedenen Compilern beanspruchte Laufzeit zum Kompilieren von PHOENIX ist in Abbildung 4.2 dargestellt.

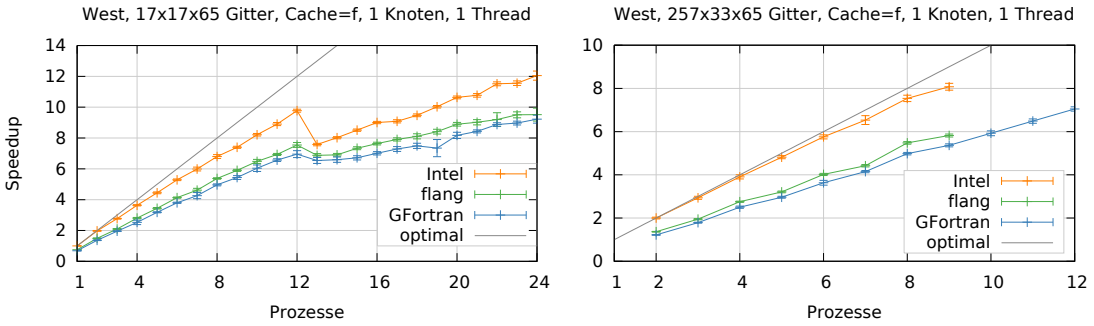

Abbildung 4.1.: Ergebnisse verschiedener Compiler
Links ist die Messung mit dem kleinen und rechts mit dem großen Gitter zu sehen. Variiert wird die Zahl der MPI Prozesse bei konstanter Anzahl Threads pro Prozess.

[1] GCC Optimization: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

[2] Das Limit hat vermutlich mit der Allokation von Structs und Arrays zu tun. Die für den Intel Compiler verwendete PHOENIX-Architektur `X86_64-INTEL-F95-V17-MPI-SMP` setzt allerdings das empfohlene Flag `-heap-arrays` , siehe: `https://software.intel.com/en-us/articles/intel-fortran-compiler-increased-stack-usage-of-80-or-higher-compilers-causes-segmentation-fault`

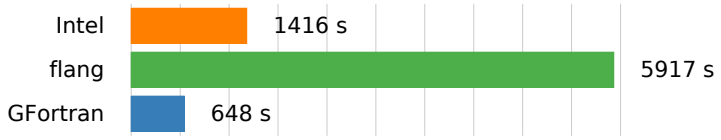

Abbildung 4.2.: Compilier-Laufzeiten im Vergleich

Alle Varianten wurden mit `./mk_phoenix -j16 <Zielarchitektur>` bei vorhergehendem `./mk_clean` auf dem Host magny1 des Arbeitsbereichs Wissenschaftliches Rechnen ausgeführt. Der Einsatz der Compiler mit oder ohne Unterstützung von OpenMP führte zu etwa gleichen Laufzeiten.

4.1.2. Anmerkungen zu flang und LLVM

Leider konnte flang erst mit Stand des Commits 45d7aeb vom 07.07.2018 verwendet werden. Zuvor verhinderten Probleme[1] das Kompilieren von flang selbst. Da genanntes Datum zu nah am Ende des Zeitrahmens dieser Arbeit liegt, kann hier nicht weiter auf flang eingegangen werden. Die Evaluation, dass sich PHOENIX überhaupt ohne umfangreiche Anpassungen mit flang kompilieren lässt, kann eine Grundlage für zukünftige Arbeiten darstellen.

Flang wurde anfänglich ausgewählt, weil das LLVM Framework interessante Ansätze bietet, damit Codeanalyse zu betreiben. Zusätzliche Tranformations-, Optimierungs- oder Analyse-Schritte während des Kompilierens, sogenannte Passes, können selbst entwickelt und dem Compiler hinzugefügt werden [LLV18]. Als Zwischenergebnis des Kompilationsprozesses steht die intermediate representation (IR) eines Programms solch einem Pass zur Verfügung. Als solche ist sie mit mehr Informationen angereichert und zur automatisierten Analyse deutlich zugänglicher als der Quelltext. Entsprechende Kenntnisse sind im Arbeitsbereich Wissenschaftliches Rechnen bereits vorhanden. [Röd17, Ble17].

4.1.3. Vergleich gemeinsamer und verteilter Speicher

Nach dem Vergleich der Compiler, soll nun das Augenmerk auf PHOENIX selbst gerichtet werden. Um einen Überblick über das Skalierungsverhalten zu gewinnen, werden zunächst einige kurze Testläufe betrachtet, wozu das kleine $17 \times 17 \times 65$ Gitter verwendet wird.

Der Einsatz von OpenMP als Methode des gemeinsamen Speicherzugriffs und MPI für den verteilten Zugriff in PHOENIX wurde bereits im Abschnitt 2.1 erläutert. Um einen Eindruck der durch MPI erzielten Leistung zu erhalten, wird in einer Messreihe die Zahl der OpenMP Threads auf 1 festgehalten und sukzessive die Zahl der MPI Prozesse erhöht. Analog wird für OpenMP verfahren.

Hierbei traten Probleme beim Einsatz des GFortran Compilers zusammen mit OpenMP auf. PHOENIX lieferte damit zwar Ergebnisse und diese werden auch zur Bewertung der OpenMP-Leistung herangezogen, aber das so erzeugte Programm zeigt Probleme beim Einsatz von MPI. Statt wie erhofft ein einziges Kompilat zum Vergleich beider Techniken heranzuziehen, werden hier zunächst zwei verschiedene GFortran Varianten gezeigt. Die ohne OpenMP-Support

[1] flang Issue #434: `https://github.com/flang-compiler/flang/issues/434`

kompilierte Variante wird für die Bewertung der OpenMP-Leistung gar nicht betrachtet, da diese hier konstant sein sollte (der Parameter wird schlicht ignoriert).

Die Ergebnisse in Abbildung 4.3 zeigen eine gute Skalierung mit steigender Zahl der MPI Prozesse, die im hier betrachteten unteren Ende dem optimalen Verlauf recht nahe kommt. Werden mehr Prozesse als vorhandene Kerne gestartet, bricht die Leistung etwas ein, reizt man das Hyper-Threading jedoch voll aus (auf beiden Systemen also 2 Prozesse pro Kern), kann insgesamt die Laufzeit weiter reduziert werden. Durch Erhöhung der OpenMP Threads lässt sich jedoch kaum ein Leistungsgewinn erzielen. Bei einem Thread pro Kern und auch bei voll ausgereiztem Hyper-Threading wird nur ein Speedup von etwa 2 erreicht.

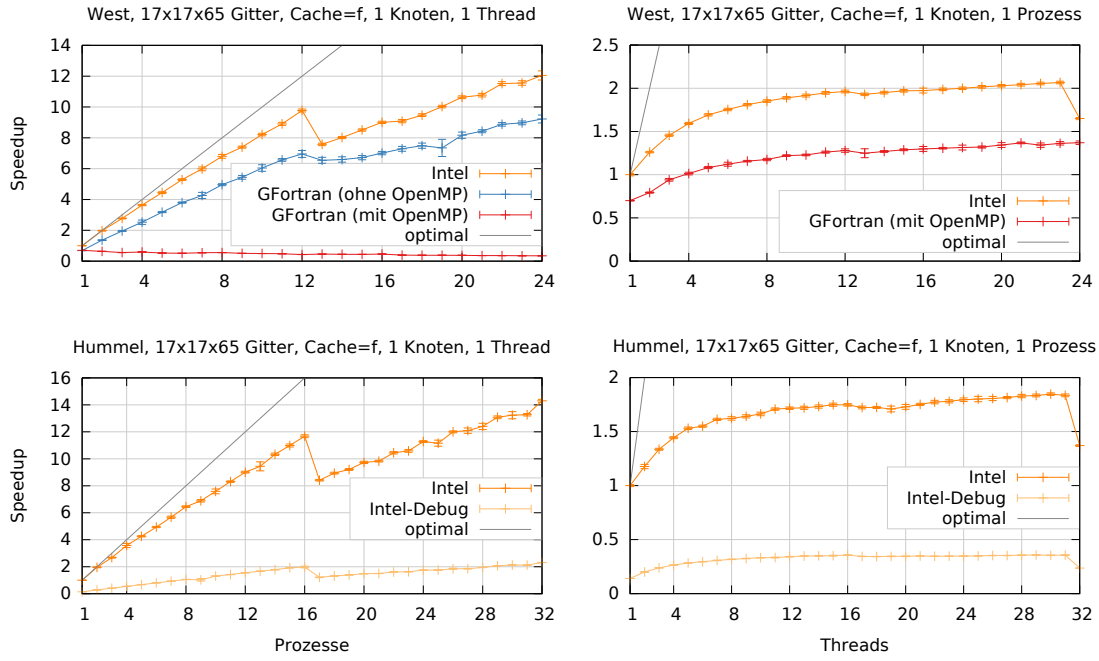

Abbildung 4.3.: Vergleich Prozesse (MPI) und Threads (OpenMP)

Links wurden bei je einem Thread die Zahl der MPI Prozesse erhöht, rechts dagegen für einen Prozess die Zahl der OpenMP Threads. Oben ist das West System zu sehen, unten Hummel. Die OpenMP Variante mit GFortran lief erfolgreich, versagt aber in der MPI Performance, wie man oben links sieht. Bei der Prozesszahl kann man gut am Einbruch oberhalb von 12 bzw. 16 Prozessen erkennen, wann die Anzahl Kerne ausgereizt ist und das Hyper-Threading einsetzt. Referenz für den Speedup 1 ist die jeweils schnellste Variante bei einem Prozess und einem Thread, hier die Intel-Variante mit einer Laufzeit von etwa 4 Minuten auf West und etwa 2,5 Minuten auf Hummel.

Die zuvor gezeigten Messreihen entsprechen allerdings keiner sinnvollen Aufteilung, da stets eine Technik bewusst blockiert wird. Für eine effektive Zusammenarbeit sollte daher eine geeignete Kombination von mehreren Threads innerhalb vieler Prozesse festgelegt werden. [HW10]

Um PHOENIX hinsichtlich einer solchen Kombination zu untersuchen wurden weitere Messreihen über die Prozesszahlen aufgenommen, bei der jeweils eine konstante, aber von 1

verschiedene Anzahl Threads festgelegt wurde. Die Ergebnisse im Vergleich zur Messreihe mit einem Thread sind in Abbildung 4.4 dargestellt.

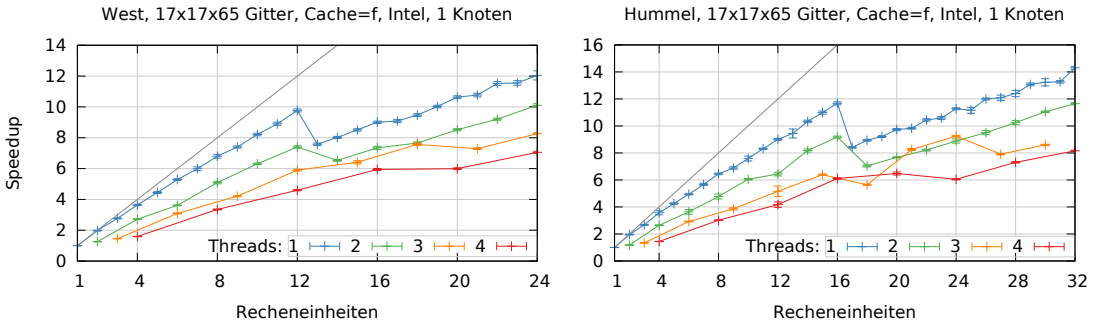

Abbildung 4.4.: Variation der Anzahl Threads pro Prozess
Links ist die Messung auf dem West und rechts die auf dem Hummel Cluster zu sehen. Die genannten Recheneinheiten sind das Produkt aus der Prozess- und Thread-Anzahl. Insgesamt scheint ein Kern effizienter in einen Prozess als einen Thread investiert zu sein.

Die Ergebnisse dieser Messungen zeigen aber ebenfalls, dass sich in diesem Fall der Einsatz von OpenMP Threads nicht lohnt. Man ist besser beraten, die CPU komplett durch MPI Prozesse auszulasten und dies bis zur vollen Auslastung des Hyper-Threadings.

Hier sei aber angemerkt, dass die OpenMP Leistung auf Many Integrated Core Architekturen wie dem Intel Xeon Phi[1] besser ist und OpenMP daher durchaus sinnvoll mit PHOENIX verwendet werden kann. Ein solches System wurde im Rahmen dieser Arbeit kurz evaluiert, wird aber im Weiteren nicht betrachtet, hauptsächlich weil kein Cluster mit mehreren identischen Systemen zur Verfügung stand und die Xeon Phi Reihe vom Hersteller auch nicht weiter fortgeführt wird.[2]

Im Allgemeinen erfordert die Vereinigung von MPI und OpenMP in einer Hybrid-Parallelisierung zusätzliche Planung. Es müssen Ebenen definiert werden, auf der die Techniken zum Einsatz kommen, da sie nicht simpel zu vereinen sind. Z.B. kann OpenMP auf innere Schleifen angewendet werden, mittels MPI hingegen wird außerhalb davon die Arbeit aufgeteilt. [HW10]

Die in PHOENIX verwendeten Ebenen wurden in Abschnitt 2.1 beschrieben. Der Aufruf des Trackers erfolgt in einer Schleife, die per MPI parallelisiert ist. Das Innere des Trackers verwendet dagegen OpenMP. Das gezeigte Verhalten von OpenMP ist daher als Hinweis zu sehen, den Code des Trackers näher zu betrachten.

Für die Leistungsanalyse wird im Folgenden die OpenMP Skalierung nicht weiter betrachtet werden.

[1] Intel Xeon Phi: `https://www.intel.com/content/www/us/en/products/processors/xeon-phi.html`

[2] Intel lässt die Technologien wie AVX-512 in moderne CPUs einfließen und erhöht (im Vergleich zum Xeon Phi nur mäßig) die Zahl vollwertiger Kerne: `https://ark.intel.com/products/series/125191/Intel-Xeon-Scalable-Processors`

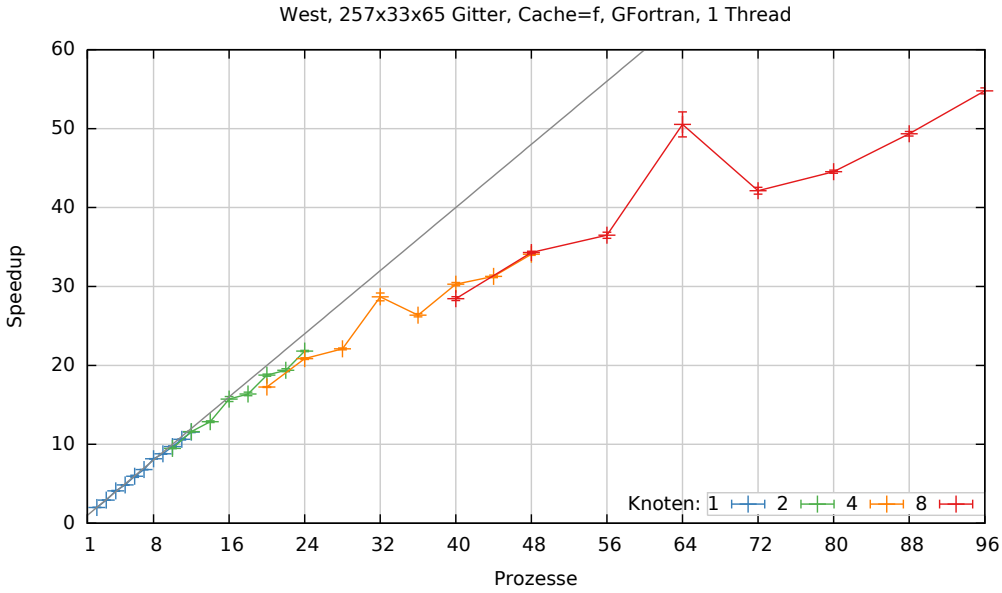

Abbildung 4.5.: Speedup über mehrere Knoten
Es ist der Speedup auf 1 bis 8 Knoten für Berechnungen auf dem großen 257×33×65 Gitter dargestellt. Pro Knoten wurden bis zu 12 Prozesse gestartet, aufgetragen ist die Summe über alle Knoten. Die bei 2 Prozessen gemessene Laufzeit beträgt etwa 6 Stunden, als Referenz für den Speedup 1 gelten daher etwa 12 Stunden. Die Spitzen bei 32 und 64 ergeben sich vermutlich, weil sich hier die 16×16 Raumwinkel glatt aufteilen lassen.

4.1.4. Knotenübergreifende Parallelisierung

Wichtig für den Einsatz im Hochleistungsrechnen ist vor allem die Skalierung über eine Vielzahl von Knoten. Nur durch viele Knoten wird insgesamt eine immense Zahl von Prozessen erreicht und die parallele Ausführung geeigneter Programme stark beschleunigt. Im Gegensatz zum Betrieb auf einem einzelnen Knoten, spielt dann das Netzwerk für die Kommunikation der Prozesse untereinander eine entscheidende Rolle. Die Laufzeiten der Kommunikationsaufrufe werden entsprechend verlängert, wenn beteiligte Prozesse auf unterschiedlichen Knoten lokalisiert sind. Nun soll daher einmal das Verhalten von PHOENIX betrachtet werden, wenn die Zahl der Knoten gesteigert wird.

In Abbildung 4.5 ist zu sehen, wie die näherungsweise lineare Skalierung auf einem Knoten auch über mehrere Knoten fortgesetzt wird. Mit der Zahl der Knoten steigt der Verwaltungs- und Kommunikationsaufwand, was hier zu einer gewissen Reduktion des Speedups führt. Die Skalierung entspricht der Erwartung aufgrund der vorherigen Betrachtungen und wird daher als gut bewertet.

4.1.5. Verwendung des Tracking Cache

Im Abschnitt 2.3 wurde beschrieben, dass die Option besteht, PHOENIX mit einem Cache zu betreiben, der die Geometrie der Charakteristiken über die Iterationen hinweg vorhält.

Wie beschrieben steigt dadurch der Bedarf an Hauptspeicher stark an. In Abbildung 4.6 wird der Speedup durch Verwendung dieses Caches gezeigt. Die eingesetzten Kosten bewirken also durchaus einen deutlichen Gewinn an Leistung. Sofern ausreichend RAM vorhanden ist, empfiehlt sich also der Einsatz.

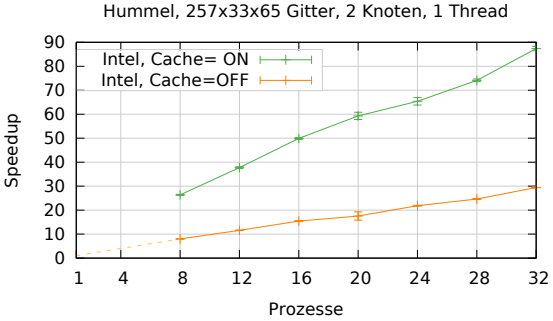

Abbildung 4.6.: Speedup durch Tracking Cache
Hier gilt als Referenzwert für den Speedup 1 der ohne Cache geschätzte Wert bei einem Prozess.

4.1.6. Begrenzende Faktoren

Wie schon zuvor in Abschnitt 4.1.1 erläutert, kann es vorkommen, dass der RAM eines Knoten nicht ausreicht, um all seine Kerne für MPI Prozesse zu verwenden. Mit steigender Zahl der Prozesse, sollen ja nach dem Parallelisierungsschema gemäß Kapitel 2, weitere Raumwinkel (θ, ϕ) zeitgleich auf einem Knoten berechnet werden. Jeder Prozess benötigt dann genügend RAM, um das komplette sphärische Gitter und alle hindurchgehenden Charakteristiken darzustellen. Aus der Größe des Gitters und der Zahl der Charakteristiken in Relation zum verfügbaren RAM des Knotens, ergibt sich daher ein Maximalwert für die Prozesse pro Knoten.

Ein weiteres Limit ergibt sich direkt aus dem Parallelisierungsschema. In erster Ebene werden die Wellenlängen auf Prozessgruppen aufgeteilt, innerhalb jeder Gruppe werden dann die Raumwinkel aufgeteilt. Eine Gruppe kann daher nur so viele Prozesse umfassen, wie Raumwinkel zur Verteilung anstehen. Ebenso kann es nur so viele Gruppen geben, wie Wellenlängen vorgesehen sind. Die gewünschten Auflösungen geben also eine Maximalzahl an Prozessen vor. Zusammen mit der zuvor beschriebenen möglichen Zahl der Prozesse pro Knoten, ergibt sich daraus ein Maximum an verwendbaren Knoten.

Natürlich ist es leicht, weitere Hardware sinnvoll auszulasten, indem die genannten Auflösungen erhöht werden. Dies vergrößert aber den Problemumfang und wäre als Weak Scaling zu betrachten. Außerdem kann bei Einsatz der maximalen Hardware die Rechenzeit immer noch beachtlich sein, so dass man im Sinne des Strong Scaling gerne mehr Hardware zur schnelleren Lösung einer konkreten Problemgröße einsetzen würde. Vor allem ist es die Auflösung des räumlichen Gitters, die dann die Rechenzeit bestimmt. Diese möchte man aber vielleicht eher erhöhen als die Auflösung der Wellenlängen[1] und Raumwinkel. Hier gilt es also, auch technische Details in die Wahl der Auflösungen einfließen zu lassen.

4.2. Profiling

Um ein Profil von PHOENIX zu erstellen wird Score-P eingesetzt, wie bereits in Abschnitt 3.2 beschrieben worden ist. Um die Instrumentierung vorzunehmen, muss PHOENIX mit dem Compiler-Wrapper `scorep` kompiliert werden. Dazu muss erneut der Buildprozess, analog zur

[1] Die Anforderungen an das Spektrum ergeben häufig eine Zahl der Wellenlängen von 10^6 bis 10^7.

Listing 4.1: Anpassung zur Score-P Instrumentierung [Source/Compiler.options]

```
...
#X86_64-GFORTRAN-MPI-SCOREP F77=scorep mpif90 -m64 CPUOPTS
#X86_64-GFORTRAN-MPI-SCOREP F90=scorep mpif90 -m64 CPUOPTS
#X86_64-GFORTRAN-MPI-SCOREP LD=scorep mpif90 -m64 CPUOPTS
#X86_64-GFORTRAN-MPI-SCOREP CC=scorep mpicc -m64 GCC_CPUOPTS
...
```

Beschreibung in Abschnitt 2.4, angepasst werden. Zusätzlich zu den Anpassungen, die in Anhang A zu finden sind, muss der Aufruf des Compilers `mpicc` für C bzw. `mpif90` für Fortran durch `scorep mpicc` bzw. `scorep mpif90` ersetzt werden. Der angepasste Ausschnitt für die GFortran Variante, die hierzu verwendet wird, ist in Listing 4.1 zu finden.

Zum Erstellen des Profils wurde PHOENIX mit 8 Prozessen und dem kleinen 17×17×65 Gitter gestartet. Über die Umgebungsvariable `SCOREP_EXPERIMENT_DIRECTORY` wird im Jobscript angegeben, wo das Profil abgelegt werden soll. Bei der Ausführung des Programms werden dann die Daten aufgenommen. Das Profil lässt sich in Textform über den Befehl `scorep-score -r <experiment_directory>/profile.cubex` ausgeben, dies ist in Tabelle 4.1 aufbereitet. In Abbildung 4.7 wird daraus der Anteil der Funktionen an der Gesamtlaufzeit grafisch dargestellt.

Die mit der Parallelisierung verbundene Kommunikation und Synchronisation benötigt zwar viel Zeit für einen Funktionsdurchlauf, wird aber selten aufgerufen und macht daher nur einen geringen Anteil aus. Besonders auffällig sind dagegen die Funktionen `cell_coord()` und `cell_step()`, die sowohl einen großen Anteil der Laufzeit ausmachen als auch sehr häufig aufgerufen werden. Beide, sowie auch `r_to_s()` und `compute_FS_coeffs()` sind Subroutinen des Trackers.

In `cell_coord()` erfolgt die Transformation von kartesischen Kordinaten, in denen die Charakteristiken als Geraden geführt werden, hin zu Kugelkoordinaten, um die Gitterelemen-

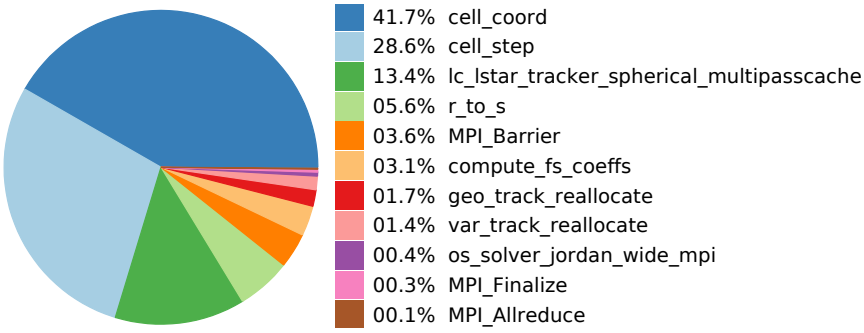

Abbildung 4.7.: Aufteilung der Laufzeit auf einzelne Funktionen
Aus den Profildaten des 17×17×65 Gitters auf 8 Prozessen wird der Anteil der gelisteten Funktionen an der Gesamtlaufzeit dargestellt.

te der jeweiligen Orte zu identifizieren. Die Funktion `cell_step()` hat die Aufgabe, eine Charakteristik gerade so weit fortzuführen, dass vom aktuellen Gitterelement der nächste Nachbar erreicht wird. Der Inhalt dieser Funktionen sollte auf Optimierungspotential geprüft werden, sowie auch die Zahl der Aufrufe nach Möglichkeit reduziert.

Tabelle 4.1.: Score-P Profil

type	visits	time[s]	time[%]	time/visit[μs]	region
ALL	2,435,888,797	856.09	100.0	0.35	ALL
USR	2,435,884,776	818.49	95.6	0.34	USR
MPI	3,469	33.68	3.9	9710.15	MPI
COM	552	3.91	0.5	7080.63	COM
USR	1,320,953,465	357.17	41.7	0.27	cell_coord
USR	430,422,330	48.34	5.6	0.11	r_to_s
USR	372,947,432	245.01	28.6	0.66	cell_step
USR	186,473,716	26.93	3.1	0.14	compute_fs_coeffs
USR	62,516,480	11.86	1.4	0.19	var_track_reallocate
USR	62,516,480	14.28	1.7	0.23	geo_track_reallocate
MPI	3,000	0.61	0.1	203.27	MPI_Allreduce
USR	3,328	114.31	13.4	34348.38	lc_lstar_tracker_spherical_multipasscache
MPI	112	30.41	3.6	271504.64	MPI_Barrier
COM	104	3.46	0.4	33282.96	os_solver_jordan_wide_mpi
MPI	8	2.41	0.3	300972.22	MPI_Finalize

Der obere Abschnitt liefert eine Zusammenfassung des gesamten Laufs (ALL) sowie der Anwendung (USR), der Parallelisierung (MPI) und gesondert der MPI Kommunikation (COM). Im unteren Abschnitt werden einzelne Funktionen mit der Zahl ihrer Aufrufe und der Verweilzeit aufgelistet. Ausgelassen wurden alle Datensätze, die weniger als 0,1% der Laufzeit ausmachen. Der Tracker selbst wurde in 13 Iterationen für alle 16×16 Raumwinkel = 3328 mal aufgerufen.

4.3. Tracing

Zusätzlich zum Profiling wird nun auch mittels Tracing ein Blick auf die Abläufe in PHOENIX geworfen. Analog zum Profiling basiert die Datenaufnahme auf Score-P. Als zusätzlicher Schritt ist vor Programmstart die Variable `SCOREP_ENABLE_TRACING=true` zu setzen, damit alle Funktionsaufrufe, mit Zeitstempel versehen, als Ereignis gespeichert werden. Diese werden im RAM gesammelt, um die dadurch verursachte Leistungsminderung in Grenzen zu halten. Da sehr viele Daten anfallen und der RAM beschränkt ist, wird hier ein Minimalbeispiel betrachtet mit 3×3 Raumwinkeln, einem 5×5×5 Gitter und dem Solver begrenzt auf 5 Iterationen.

Um die im Profiling gewonnenen Informationen weiter anzureichern, lässt sich in Vampir der aus den Ereignissen ermittelte Call Tree darstellen, wie in Abbildung 4.8 gezeigt. Hier ist zu sehen, dass die auffällige Funktion `cell_coord()` sowohl direkt aus dem Tracker als auch aus der anderen auffälligen Funktion `cell_step()` aufgerufen wird. Der hohe Aufwand letzterer könnte also ein Folgeeffekt sein.

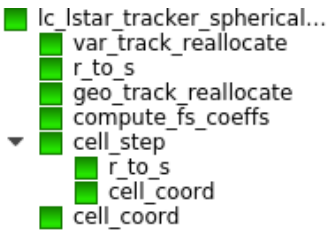

Abbildung 4.8.: Call Tree

Der Fokus des Tracings liegt aber auf der Darstellung der Ereignisse auf einer Zeitachse. In Abbildung 4.9 ist die Visualisierung der Spurdaten in Vampir zu sehen. Hier werden die zuvor identifizierten Funktionen `cell_coord()` und `cell_step()` während einer einzigen Iteration gezeigt. Wie schon aus dem Profil bekannt, sind dies viele Aufrufe, die in Summe einen großen Anteil der Laufzeit ausmachen. Im Tracing wird gezeigt, dass sich die Aufrufe divers über die Laufzeit des Trackers verteilen.

Betrachtet man alle 5 Iterationen des Minimalbeispiels nebeneinander fällt zunächst auf, dass die erste mehr Zeit beansprucht. Hier sind Initialisierungen enthalten, die in folgenden Iterationen entfallen. Weiter fallen lange Wartezeiten bei der Synchronisation in der `MPI_Barrier()` auf, die aber nur manche Prozesse betreffen. Hier ist wichtig festzustellen, dass sich diese Wartezeiten nicht statistisch über alle Prozesse verteilen, sondern dass stets dieselben Prozesse nur etwa halb solange rechnen wie andere und dann warten müssen.

Das Minimalbeispiel betont hier einen Effekt, der in realen Programmläufen wahrscheinlich weniger stark auffällt, denn hier hat jeder Prozess nur genau einen Raumwinkel (θ, ϕ) zu betrachten. Offensichtlich hängt die Rechenzeit des Trackers von diesem Raumwinkel ab. Wie stark dieser Effekt bei größeren Auflösungen auftritt, wird von dieser Abhängigkeit bestimmt sein. Diese soll daher im Weiteren untersucht werden.

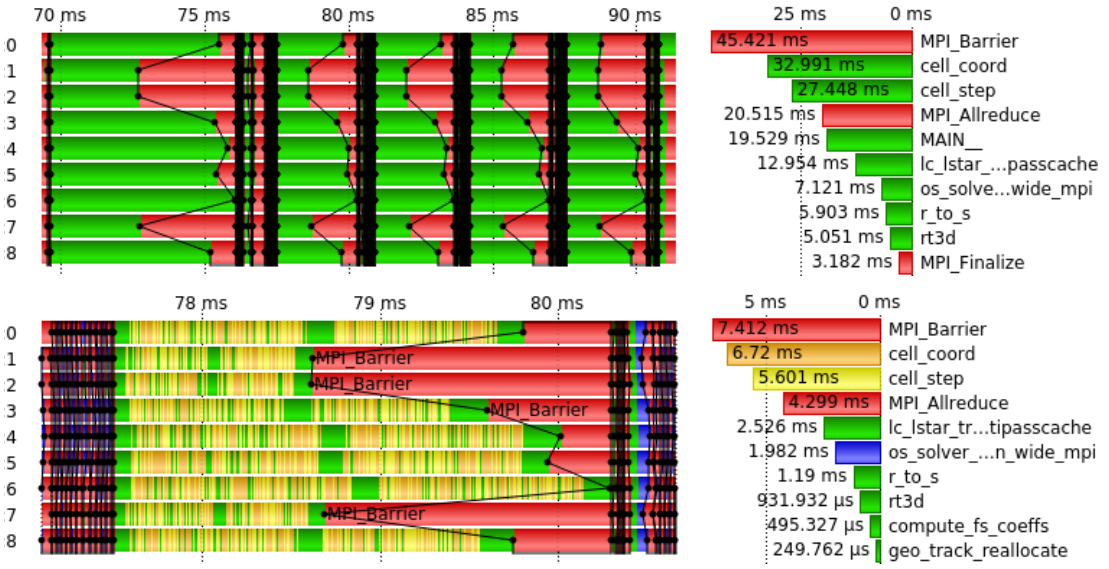

Abbildung 4.9.: Vampir Tracing

Oben sind die 5 Iterationen zu sehen, wobei die erste aufgrund von Initialisierungen länger ausfällt. Rot markiert die MPI Routinen, die eigentliche Anwendung ist grün dargestellt. Jeder der 9 Prozesse hat in diesem Minimalbeispiel genau einen Raumwinkel zu bearbeiten. Einige Prozesse benötigen dafür in jeder Iteration weniger Zeit als andere. Unten ist die zweite Iteration herausgegriffen. Die hier blau eingefärbte Funktion zeigt, dass der OS Solver zwischen den Aufrufen des Trackers tätig ist. `cell_coord()` und `cell_step()` sind hier orange bzw. gelb dargestellt und man sieht ihre Dominanz. Ein großer Anteil der Zeit fällt hier aber auch auf Wartezeiten der MPI Barrier aufgrund der ungleich verteilten Last.

4.4. Lastverteilung

In der Analyse des Tracings wurde herausgestellt, dass Prozesse, abhängig vom zu betrachtenden Raumwinkel (θ, ϕ), unterschiedliche Laufzeiten zur Berechnung einer Iteration zeigen. Die Aufteilung der Raumwinkel auf alle verfügbaren Prozesse erfolgt aber schlicht nach deren Anzahl und ist nicht nach deren Komplexität gewichtet. Wie sich die Komplexität eines Raumwinkels im Vergleich zu anderen darstellt, ist auch gar nicht explizit bekannt. Im Folgenden soll die Abhängigkeit einmal betrachtet werden.

4.4.1. Anzahl der Charakteristiken

Eine der Aufgaben des Trackers, wie in Abschnitt 2.3 beschrieben, ist das Erzeugen einer geeigneten Menge von Charakteristiken, so dass alle Gitterelemente von mindestens einer Charakteristik durchlaufen werden. Für jeden Raumwinkel muss das sphärische Gitter von einer anderen Menge Charakteristiken durchlaufen werden, die sich mindestens in ihren Startpunkten unterscheidet. Hier liegt die Vermutung nahe, dass sich auch ihre Anzahl unterscheidet.

Mit steigender Anzahl der Charakteristiken nimmt nicht nur der Aufwand dieser Phase des Geometrie-Aufbaus zu, sondern die Charakteristiken müssen auch anschließend alle durch das sphärische Gitter verfolgt werden. Die Anzahl der Charakteristiken ist daher ein Maß für den Aufwand eines Tracker-Aufrufs.

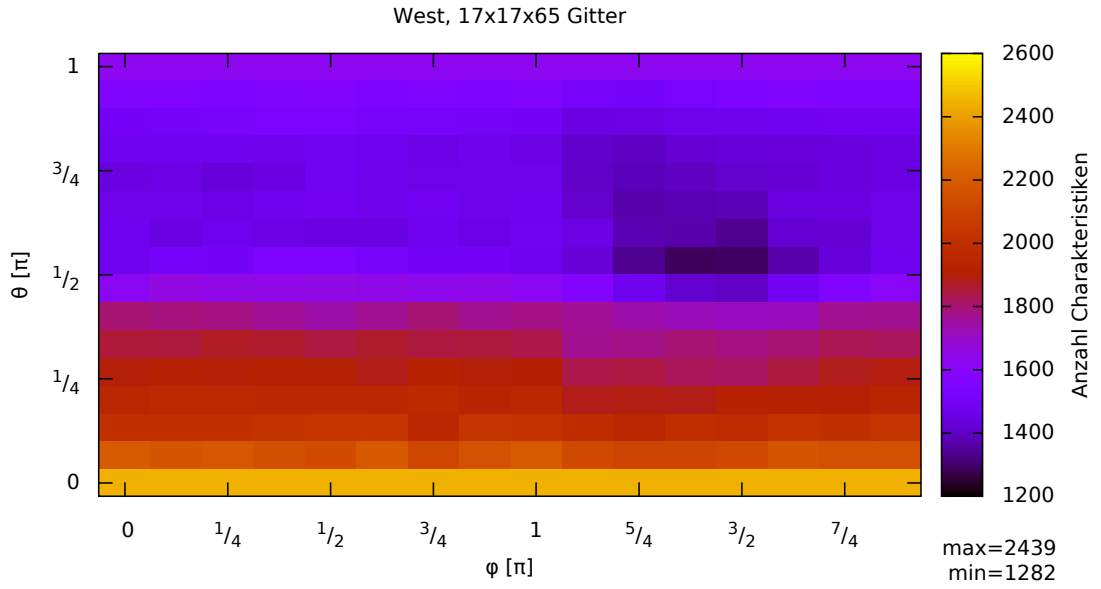

Abbildung 4.10.: Abhängigkeit der Anzahl Charakteristiken vom Raumwinkel
Der Farbverlauf zeigt für jeden Raumwinkel (θ, ϕ) die Zahl der benötigten Charakteristiken an. Das Minimum von 1282 Charakteristiken wird bei etwa $\theta = \frac{1}{2}\pi, \phi = \frac{3}{2}\pi$ angenommen, das Maximum von 2439 am unteren Rand bei $\theta = 0$. Es scheint $\theta > \frac{1}{2}\pi$ vorteilhaft zu sein, ϕ bis auf den Bereich nahe des Minimums eher unbedeutend.

Um die Anzahl der Charakteristiken in Abhängigkeit zum Raumwinkel quantitativ zu erfassen, wird PHOENIX um die Protokollierung dieser Anzahl `n_chars` erweitert, so wie es in Anhang B beschrieben ist. Der Tracker wird ohnehin für jeden Raumwinkel einmal pro Iteration aufgerufen, so dass der Raumwinkel `theta, phi` und die Anzahl `n_chars` zu einem Datensatz zusammengefügt werden. Wie in Abschnitt 2.3 bei der Beschreibung des Tracking Caches erwähnt worden ist, wird in jeder Iteration dieselbe Menge Charakteristiken erzeugt (oder wiederverwendet), so dass hier keine Abhängigkeit zur Iteration zu erwarten ist. Zur Datenaufnahme kann also eine beliebige Iteration herausgegriffen oder die Zahl der Iterationen sogar für diesen Programmlauf mittels `itlamb = 1` auf eine einzelne beschränkt werden. Die Daten werden bei Verwendung des kleinen Gitters aufgenommen.

In Abbildung 4.10 wird für die 16×16 Raumwinkel die jeweils benötigte Zahl der Charakteristiken dargestellt. Es fällt auf, dass das Maximum hier etwa doppelt so groß ist wie das Minimum. Die Spanne ist also sehr groß, was bei entsprechend ungünstiger Aufteilung der Raumwinkel auf die Prozesse, zu ungleich verteilter Last führen kann.

Ebenso fallen zwei Bereiche des Diagramms ins Auge. Für $\theta \geq \frac{\pi}{2}$ liegen niedrige Werte vor, in denen sich ein kleiner Bereich als Minimum darstellt. Für $\theta < \frac{\pi}{2}$ zeigt sich ein starker Anstieg bis zum Maximum bei $\theta = 0$. Bis auf die genaue Lage des Minimums scheint ϕ keine bedeutende Rolle zu spielen. Die zu verteilende Last ist also hauptsächlich von θ abhängig.

4.4.2. Lastausgleich

Zur Aufteilung der Last auf die verfügbaren Prozesse bedeutet die Erkenntnis des vorigen Abschnitts, dass nach Möglichkeit ein Prozess mehrere θ Werte zur Bearbeitung erhalten sollte und ggf. ϕ gleich bleibt. Denn würde man dagegen den Prozess an ein θ binden und mehrere ϕ zuweisen, so würde man einem Prozess nur viele kleine, einem anderen Prozess aber nur viele große Aufgaben zuweisen. Bei der Verteilung der θ besteht die hohe Wahrscheinlichkeit, dass sich kurze und lange Aufgaben abwechseln und statistisch ausgleichen.

Der Abschnitt in PHOENIX, der die Raumwinkel verteilt, ist in Listing 4.2 zusammengefasst. Die innere Schleife iteriert über ϕ und es wird mittels des Modulo-Operators reihum den

Listing 4.2: Verteilung der Raumwinkel (θ, ϕ) auf die Prozesse [`Source/3DRT/3D.for`]

```
276     do i_theta=1,ntheta
277      ...
281      theta = ...
282      ...
286      do i_phi=1,nphi
287       ...
288       phi = ...
289       ...
293       MPI_counter = MPI_counter+1
294       if(mod(MPI_counter,FS_size) .ne. FS_rank) cycle
295       ...
425       call LC_Lstar_solver(pve_grid,theta,phi,BC_set)
```

Prozessen ein Raumwinkel zugewiesen. Dadurch bekommt nach Möglichkeit jeder Prozess einen Anteil des gerade gültigen θ, also Anteile zu etwa gleichem Schwierigkeitsgrad. Im Extrembeispiel, wenn jeder Prozess genau zwei Raumwinkel erhält, wird es also einer aus dem unteren Bereich bei (θ_i, ϕ_j) und einer aus dem oberen bei $(\theta_i + \frac{\pi}{2}, \phi_j)$ sein. Dies ist zumindest keine schlechte Verteilung. Besser ginge es nur mit aufwendigen Algorithmen, die detailliertere Kenntnis über die genaue Verteilung voraussetzen, die an dieser Stelle des Programms gar nicht vorliegt.

4.4.3. Charakteristiken pro Gitterelement

Um der Ursache des in Abbildung 4.10 gezeigten Verlaufs näher zu kommen, soll nun einmal untersucht werden, wie sich die Charakteristiken auf das Gitter verteilen. Zuvor wurde die Gesamtzahl der Charakteristiken betrachtet, nun soll einmal der Fokus auf die Zahl der Charakteristiken gelegt werden, die ein einzelnes Gitterelement durchlaufen. Ist insgesamt eine hohe Anzahl nötig, so stellt sich die Frage, ob sich die zusätzlichen Charakteristiken über das ganze Gitter verteilen oder auf einzelne Elemente verdichten. Vielleicht treten auch Muster oder zusammenhängende Bereiche von dicht durchströmten Elementen auf, die Einsichten zum Aufbau liefern.

Das gesamte Gitter stellt in jedem Zustand bereits ein komplexes Konstrukt dar. Zur Betrachtung sollen daher hier lediglich zwei einzelne Zustände bei gewählten Raumwinkeln der Charakteristiken genügen. Als Raumwinkel werden das in Abbildung 4.10 identifizierte Minimum und ein ungünstiger Wert bei $\theta = 0.52, \phi = \pi$ festgelegt.

In PHOENIX wird im Array `chars_pv` für jedes einzelne Gitterelement `(ir,it,ip)` die Information abgelegt, von wie vielen Charakteristiken es durchlaufen wird. Um diese Daten zu erheben wird analog zu Abschnitt 4.4.1 vorgegangen. Die an PHOENIX vorgenommenen Anpassungen sind in Anhang B beschrieben.

In Abbildung 4.11 werden die aufgenommenen Daten dargestellt. Das Gitter wird dabei in Schichten verschiedener Radien von außen nach innen abgetragen, wobei jedoch nicht alle 17 Schichten dargestellt sind[1].

Zu erkennen ist, dass das jeweilige Maximum auf der äußersten Schicht angenommen wird und aus der Menge aller Elemente eher heraussticht. Die Zahl der zusätzlichen Charakteristiken kommt also nicht durch sehr viele Werte nahe des Maximums zustande. Allerdings zieht sich von außen nach innen ein Strahl von Elementen mit abnehmender, aber hoher Dichte, der in seiner Richtung nahe an der der Charakteristik liegt. Als weiteres verdichtetes Gebiet fällt die gesamte innerste Schicht auf. Hier gilt zu beachten, wie schon in Abschnitt 2.3 beschrieben, dass die Elemente weiter im Inneren des Gitters i.d.R. ein im Vergleich kleineres Volumen haben als äußere Elemente[2]. Daher dürfte es in diesem Bereich schwieriger sein, viele Elemente mit wenigen Charakteristiken zu treffen.

[1] Auf der beigefügten CD in `Plots/small_west_characteristics_pve_<Raumwinkel>_anim_original.gif` werden alle Schichten animiert dargestellt.

[2] Da die Radien der einzelnen Schichten frei wählbar sind, kann man natürlich ein Gegenbeispiel konstruieren, das aber nicht dem Regelfall entsprechen wird.

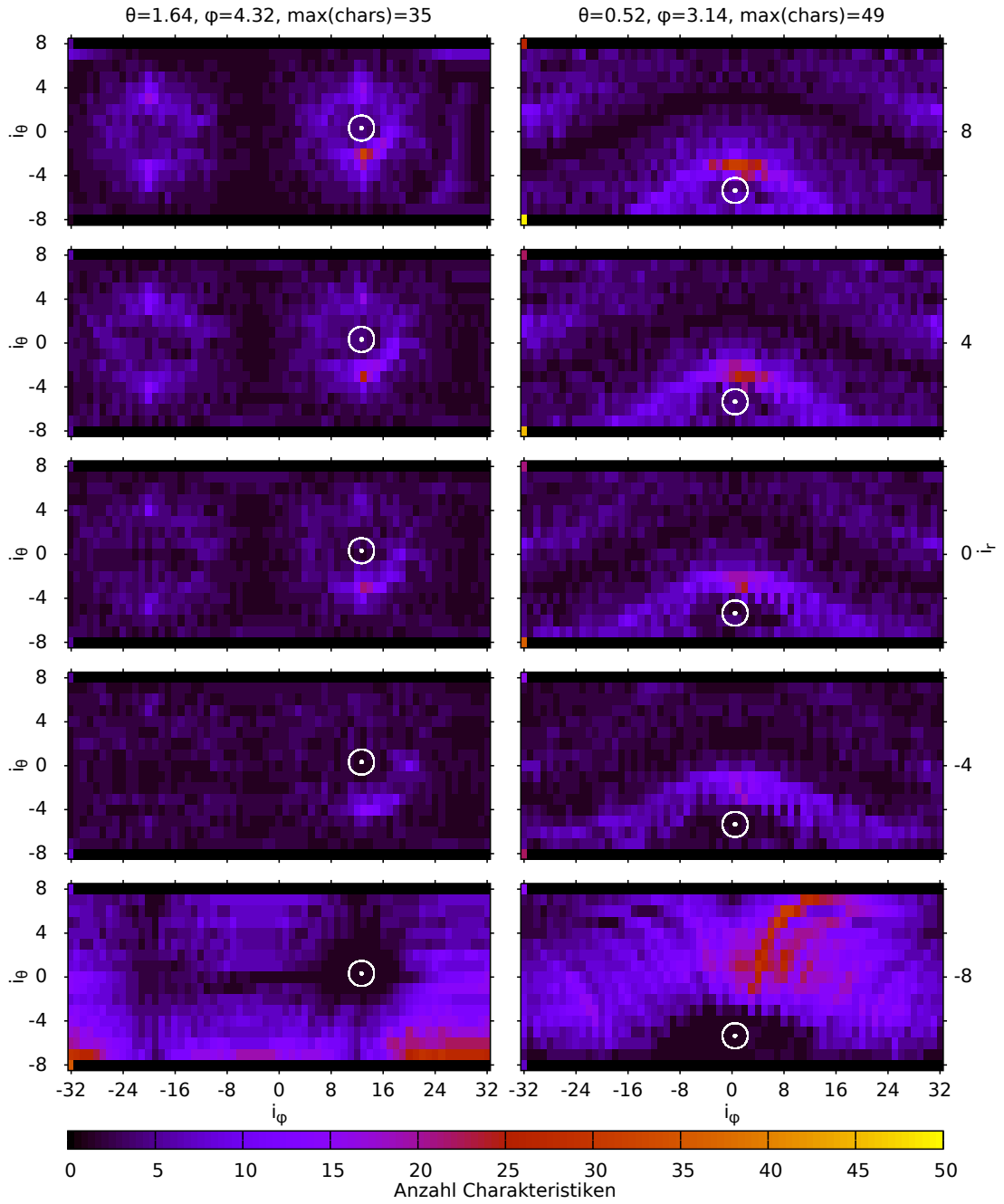

Abbildung 4.11.: Anzahl Charakteristiken pro Gitterelement

Die Markierungen zeigen die Austrittsrichtung einer durch den Mittelpunkt gehenden Charakteristik, bei gewähltem Raumwinkel. Links wurde das Minimum aus Abbildung 4.10 gewählt, rechts ein eher ungünstiger Fall bei kleinem θ. Vertikal werden von oben (außen) nach unten (innen) mehrere Schichten in i_r angezeigt. Jedes einzelne Diagramm zeigt im Farbverlauf die Anzahl der Charakteristiken, die das einzelne Gitterelement bei (i_r, i_θ, i_ϕ) durchlaufen. Man sieht wie sich, nahe des Austrittswinkels lokalisiert, eine Verdichtung durch alle Schichten zieht. Das Maximum ist jeweils außen zu finden (rechts im "Südpol" bei $i_\theta = -8$; hier ist i_ϕ entartet und daher wird nur $i_\phi = -32$ verwendet). In der innersten Schicht liegt jedoch im Mittel über alle i_θ, i_ϕ eine deutlich höhere Dichte vor als in allen äußeren.

Zielen also viele erzeugte Charakteristiken auf innerste Schichten, so werden sie auf ihrem Weg ja stets durch äußere Schichten geführt werden, die dann tendenziell Elemente mit größerem Volumen aufweisen. Die Zahl der auf dem Weg liegenden Gitterelemente wird also nach außen abnehmen, was bedeutet, dass jedes Element von zunehmend mehr Charakteristiken durchlaufen wird. Der Weg durch das Innerste ist aber durch den gegebenen Raumwinkel bedingt, so dass dies den Strahl der Verdichtung erklären könnte.

Diese Betrachtungen sollen zeigen, dass die Methode, nach der die Charakteristiken ausgewählt werden, um schließlich alle Gitterelemente zu erreichen, eine entscheidende Rolle spielt. Ein wichtiger Bestandteil der Methode wird die Auswahl und Reihenfolge von Gitterelementen sein, auf die erstellte Charakteristiken zielen. Aus der Summe dabei entstehender, möglicherweise vermeidbarer, Überschneidungen ergibt sich eine hohe Anzahl Charakteristiken, die es dann alle von einem Element zum nächsten durch das Gitter zu führen gilt. Dies macht einen beachtlichen Teil des Zeitaufwands aus und es würde sich lohnen, durch verbesserte Methoden, die Zahl der Charakteristiken zu verringern.

4.5. Bewertung

PHOENIX zeigt eine gute Skalierung über die Zahl der MPI Prozesse, auch über mehrere Knoten hinweg. Die Programmteile, in denen die MPI Aufteilung stattfindet und die den sphärischen Tracker umschließen, benötigen im Rahmen dieser Arbeit keine besondere Aufmerksamkeit. Auch aus Sicht der Lastverteilung, wobei die Last wie gezeigt hauptsächlich vom Winkel θ der Charakteristik abhängt, liegt kein akuter Bedarf vor.

Eine schlechte Skalierung mittels OpenMP, das innerhalb des Trackers verwendet wird, sowie die Ergebnisse des Profilings, legen nahe, im Quellcode des Trackers nach Verbesserungsmöglichkeiten zu suchen. Hierfür kommen vor allem die Funktionen `cell_coord()` und `cell_step()` inhaltlich in Frage. Hilfreich wäre auch die Zahl ihrer Aufrufe zu reduzieren. Die ständige Transformation von kartesischen in sphärische Koordinaten scheint allerdings ein inhärentes Problem der Aufgabenstellung dieses Trackers zu sein.

Ein Ansatz, um die Last und damit auch die Zahl obiger Aufrufe, zu reduzieren oder wenigstens gleichmäßiger zu verteilen, liegt darin, den Algorithmus anzupassen, der die Charakteristiken erzeugt. Hier wurde eine Abhängigkeit zum betrachteten Raumwinkel festgestellt, die es dabei zu berücksichtigen gilt. Ein entscheidender Aspekt dabei wird sein, die Menge und Reihenfolge von Gitterelementen festzulegen, die als Ziel der erzeugten Charakteristiken verwendet werden.

Da die Zahl der Charakteristiken eine Spanne aufweist, in der das Maximum etwa das doppelte des Minimums ausmacht, besteht hier viel Potential für Verbesserungen. Auch nur eine kleine Reduktion des Maximums kann schon als Fortschritt betrachtet werden. Entsprechende Ansätze sollen daher im Folgenden betrachtet und auf ihre Auswirkung auf die Leistung getestet werden.

5. Verwandte Arbeiten

Um PHOENIX zu beschleunigen und die Möglichkeiten moderner Technologien zu nutzen, wurden in mehreren Arbeiten bereits verschiedene Aspekte betrachtet.

Der Trend im Hochleistungsrechnen setzt immer mehr auf GPGPU[1] mittels Beschleunigerkarten [TOP18]. Um auch diese Technologien nutzen zu können, ohne sich allerdings auf herstellerspezifische Ansätze festzulegen, wurde in [HB11] eine Parallelisierung mittels OpenCL[2] verwendet, ein herstellerunabhängiger Standard zur Verwendung von sowohl CPU aus auch GPGPU.

Intel verfolgte mit seinen Xeon Phi[3] Prozessoren den Ansatz der Many Core Architektur, so dass die, in ihrer Leistung eher GPUs entsprechenden, Prozessoren im System als CPU klassifiziert und als solche ohne spezielle Programmier-Paradigmen verwendbar sind. Der Einsatz dieser Technologie sowie die in MPI-3 verfügbaren RMA[4] [5] Methoden wurden in [Squ16] anhand eines anderen Trackers untersucht.

Aktuell werden in der Entwicklung an der Hamburger Sternwarte Ansätze mit OpenACC[6] und Vektorprozessoren von NEC[7] verfolgt. Dokumente hierzu liegen noch nicht vor.

Während diese Arbeiten auf Einsatz zusätzlicher Technologien bauen, wird in dieser Arbeit mehr auf die Algorithmen innerhalb des sphärischen Trackers eingegangen. Während neue Technologien im Ansatz jedem Tracker zugute kommen können, dort aber auch erst einmal implementiert werden müssen, wird diese Arbeit nur den sphärischen Tracker betreffen, dies aber als Grundlage für den Einsatz diverser Technologien.

[1] GPGPU: General Purpose Computation on Graphics Processing Unit
[2] OpenCL: `https://www.khronos.org/opencl`
[3] Intel Xeon Phi: `https://www.intel.com/content/www/us/en/products/processors/xeon-phi.html`
[4] RMA: Remote Memory Access
[5] MPI RMA: `http://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-onesided.html`
[6] OpenACC: `https://www.openacc.org`
[7] NEC Aurora: `https://uk.nec.com/en_GB/global/solutions/hpc`

6. Leistungsverbesserungen

Im Folgenden sollen die in der Leistungsanalyse in Kapitel 4 aufgezeigten Teile von PHOENIX näher untersucht werden. Durch Variation der Algorithmen und anschließendem Vergleich der erzielten Leistung mit der des Originals, wird die Effektivität der Variationen bewertet. Schließlich sollen die effektivsten Ansätze kombiniert und zur Übernahme in PHOENIX empfohlen werden.

6.1. Theoretische Vorüberlegungen

Zunächst werden die Erkenntnisse aus der Leistungsanalyse in theoretischen Überlegungen beleuchtet. Aus diesen sollen sich sinnvolle Variationen am Programmcode zur weiteren Betrachtung ergeben.

6.1.1. Mögliche Probleme beim Multithreading

In Abschnitt 4.1.3 der Leistungsanalyse wurde gezeigt, dass in PHOENIX die Thread-Parallelisierung mittels OpenMP, im Vergleich zum Nachrichtenaustausch durch MPI, das auf den Ebenen außerhalb verwendet wird, ein eher schlechtes Skalierungsverhalten zeigt. Neben gemeinsamen Schwierigkeiten, wie eine möglichst gleichmäßige Verteilung der Last zu erreichen, gilt es aber auch Besonderheiten des nebenläufigen Zugriffs auf gemeinsamen Speicher zu beachten, wie dies auch beim Multithreading der Fall ist. Der Zugriff auf gemeinsamen Speicher ist von Vorteil, wenn die Threads lesend auf geteilte Inhalte zugreifen, da nicht jeder Thread eine eigene Kopie davon vorhalten muss. Ebenso ist schreibender Zugriff effizient umsetzbar, wenn genau festgelegt ist, welcher Thread für welchen Teil der Daten zuständig ist. Soll aber ein einzelnes Datum von mehreren Threads gemeinsam verändert werden (z.B. ein Zähler hochgezählt), konkurrieren die Threads um den Zugriff. Hierzu wird das Konzept der kritischen Abschnitte, in denen sich jeweils nur ein einzelner Thread zur Zeit aufhalten darf, in OpenMP über die Direktive `omp critical` bereitgestellt. Ist die Verweilzeit in diesem kritischen Abschnitt zu hoch oder wird er von allen Threads zu häufig besucht, kann es leicht passieren, dass jeweils ein Thread diesen Bereich belegt und beinahe alle anderen darauf warten, dass er freigegeben wird, um nacheinander ebenfalls einzutreten. Dadurch ergibt sich dann unbeabsichtigt wieder ein sequentieller Ablauf des Programms. [HW10, Ope18]

Die Verwendung solcher kritischen Abschnitte sollte daher besonders aufmerksam betrachtet werden. Sollte es sich innerhalb des Abschnitts um relativ einfache Änderungen (wie z.B. einen Zähler) handeln, so stellt die Eintrittskontrolle, die über einen Mutex (mutual exclusion) realisiert wird, einen relativ großen Overhead dar. Es besteht allerdings für ein-

fache Änderungen auch die Möglichkeit spezielle Maschinenbefehle[1] [2] zu verwenden, die *Lesen* und *Schreiben* in einem Befehl umsetzen und daher atomar sind (nicht unterbrechbar). Mittels OpenMP kann man dem Compiler die Verwendung solcher Befehl über die Direktive `omp atomic` anweisen. [Ope18]

6.1.2. Erzeugung der Charakteristiken

Die Leistungsanalyse hat in Abschnitt 4.4.1 gezeigt, dass die Anzahl der notwendigen Charakteristiken vom Raumwinkel (θ, ϕ) abhängt und es dadurch zu ungleicher Lastverteilung kommen kann. Abschnitt 4.4.3 zeigt an einem Beispiel, wie sich die Zahl der Charakteristiken über das sphärische Gitter verteilt. Bestimmte Bereiche, wie ein Strahl etwa in Richtung des Raumwinkels, sowie die innerste Schicht weisen eine hohe Dichte an Charakteristiken auf. Im Abschnitt 2.3 wurde die Aufgabe des Trackers beschrieben, Charakteristiken zu erzeugen und durch das Gitter zu führen. Die Menge der Charakteristiken, die notwendig ist, damit jedes Gitterelement mindestens von einer Charakteristik durchlaufen wird, ist nicht analytisch bestimmt. Die Komplikation stellt dabei dar, dass die Charakteristiken Geraden sind und sich daher die Menge ihrer Punkte leicht in kartesischen Koordinaten beschreiben lässt, worauf PHOENIX auch zurückgreift. Die Geraden in Kugelkoordinaten zu beschreiben, so dass sie zum sphärischen Gitter passen, ist dagegen schwieriger und dieser Ansatz wird in PHOENIX nicht verfolgt.

In einem äquidistanten kartesischen Gitter kann man sich leicht vorstellen, wie der Verlauf einer Gerade von einem Gitterelement zum nächsten geführt wird. Es wird sich eine äquidistant

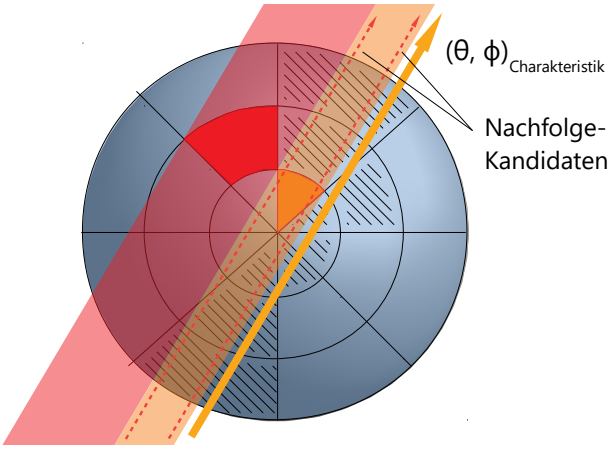

Abbildung 6.1.: Charakteristiken werden erstellt, um alle Gitterelemente zu treffen
Bei gegebener gelb eingezeichneter Charakteristik, die bereits alle schraffierten Gitterelemente besucht hat, zeigt der orange überdeckte Bereich mögliche Nachfolger, um das markierte Element zu treffen. Wählt man einen Kandidaten weit links im Bereich, werden hier deutlich mehr neue Elemente erreicht als eine Wahl weiter rechts. Der hier rot dargestellte Querschnitt weiter außen liegender Elemente ist sichtbar größer als innen. Nicht abgebildet ist die dritte Dimension des Gitters.

[1] Test-and-set: `https://en.wikipedia.org/wiki/Test-and-set`
[2] Fetch-and-add: `https://en.wikipedia.org/wiki/Fetch-and-add`

verteilte Menge bestimmen lassen, mit der dann alle Gitterelemente getroffen werden. Dagegen haben in einem sphärischen Gitter mit äquidistanter Verteilung der Radien, die innersten Elemente ein relativ kleines Volumen (in PHOENIX sind die Radien nicht äquidistant, i.d.R. werden sie aber nach innen dichter gewählt, was den Effekt noch verstärkt). Aus Sicht einer von außen auf das Gitter zielenden Charakteristik ist der Querschnitt dieser Elemente entscheidend, um sie zu treffen. Die Charakteristik, die dann einen der noch nicht durchlaufenen nächsten Nachbarn treffen soll, wird nur einen geringen Abstand zur vorigen haben. Weiter außen im Gitter wird ein größerer Abstand möglich sein. Abbildung 6.1 soll diese Problematik einmal veranschaulichen.

Weiterhin gibt es einen Spielraum, in dem möglichen Querschnitt eine neue Charakteristik zu platzieren, was die in der Abbildung gezeigten Kandidaten zeigen sollen. Verschiedene Möglichkeiten sind unterschiedlich effektiv, was die Menge an neu erreichten Gitterelementen betrifft. Die Reihenfolge, in der die Ziel-Elemente ausgewählt werden, wird ebenfalls eine Rolle spielen.

6.1.3. Asymmetrie der Raumwinkel-Abhängigkeit

Die in Abschnitt 4.4.1, Abbildung 4.10 gezeigte Abhängigkeit der benötigen Charakteristiken vom Raumwinkel (θ, ϕ) ist weder in θ noch in ϕ symmetrisch. Man würde vielleicht zunächst eine Symmetrie vermuten, da das sphärische Gitter in seiner Geometrie natürlich Symmetrien aufweist. Beim Erzeugen der Charakteristiken geht der, in Listing 6.3 gezeigte, Algorithmus allerdings alle Gitterelemente in einer bestimmten Reihenfolge durch, um die zu finden, die noch nicht getroffen wurden. Diese Reihenfolge gibt daher eine bevorzugte Richtung in θ und ϕ vor, was die Symmetrie bricht.

6.2. Angepasstes OpenMP-Schema

Das Profiling in Abschnitt 4.2 liefert uns den Hinweis, die Subroutinen `cell_coord()`, `cell_step()`, `r_to_s()` und `compute_FS_coeffs()` des Trackers näher zu betrachten. Keine dieser Funktionen ist bereits mit OpenMP parallelisiert. Da diese aber keine parallelisierbaren Schleifen enthalten, bietet sich der Einsatz von OpenMP auch nicht an. Jede der Funktionen rechnet mit einzelnen Koordinaten und ist für einen Einzelschritt des Trackings einer Charakteristik konzipiert. Hier wäre zu empfehlen, die mathematischen Methoden dieser Funktionen zu hinterfragen und nach äquivalenten Formeln zu suchen, die sich besser in den Algorithmus einfügen. Dies soll aber nicht Teil dieser Arbeit sein.

Es verbleibt der Tracker selbst, auf den bereits ein OpenMP-Schema angewendet worden ist. Aufgrund des Multi-Pass-Ansatzes gibt es wiederholte Durchläufe, in dreifach verschachtelten Schleifen, über alle Gitterelemente und, in einfachen Schleifen, über alle Charakteristiken. Die meisten dieser Schleifen sind bereits mittels `omp parallel do` Direktive parallelisiert. Eine Übersicht ist in Tabelle 6.1 gezeigt.

Das Erzeugen der Charakteristiken ist nicht parallelisiert, da dies ein sequentieller Vorgang ist. Denn die Wahl einer neuen Charakteristik hängt davon ab, welche Gitterelemente bereits von allen vorigen Charakteristiken getroffen wurden. Daher beschränkt sich dieser Durchlauf

Tabelle 6.1.: OpenMP Parallelisierungsschema des Trackers

Zeilen	Aufgabe	Iteration über	Parallelisierung
371 - 385	Speicherallokation	Gitter	omp parallel do collapse(3)
511 - 679	Erzeugung der Charakteristiken	Gitter	serielle 3-fach Schleife
723 - 737	Speicherallokation	Gitter	omp parallel do collapse(3)
754 - 768	Speicherallokation	Gitter	omp parallel do collapse(3)
787 - 928	Tracking der Charakteristiken	Charakteristiken	omp parallel do, omp atomic
1010 - 1016	Speicherung von Zwischenergebnissen	Charakteristiken	omp simd
1028 - 1042	Summe aller Charakteristiken pro Gitterelement bilden	Gitter	omp parallel do collapse(2) reduction, omp simd
1061 - 1171	Berechnungen Physik und OS Methode	Gitter, Charakteristiken pro Element	omp parallel do collapse(3), omp simd
1184 - 1434	Berechnungen OS Methode	Gitter	omp parallel do collapse(3)
1453 - 1524	Tracking der Charakteristiken	Charakteristiken	omp parallel do, omp simd (Schleife mit Schrittweite 32, darin SIMD)
1535 - 1614	Berechnungen 'finalize'	Gitter	omp parallel do collapse(2), omp simd

Übersicht über Parallelisierung der Schleifen [`Source/3DRT/LC_Lstar_tracker_spherical_MultiPassCache.for`]

auch auf das Erfassen der getroffenen Elemente, dort anfallende Berechnungen werden in späteren Durchläufen parallelisiert vorgenommen. Alle übrigen Schleifen sind mittels OpenMP parallelisiert.

Mit der OpenMP-Direktive `collapse(3)` wird der Compiler angewiesen die nachfolgende Dreifach-Schleife in eine einzelne Schleife abzuwickeln, die dann auf alle Threads verteilt wird. Der Wertebereich dieser abgewickelten Schleife ist größer als in den einzelnen Schleifen und lässt sich im Allgemeinen gleichmäßiger aufteilen. Die ursprünglichen Indizes lassen sich dabei aus dem abgewickelten Index berechnen. [HW10, Ope18]

Auch die, im Abschnitt 2.1 bereits angesprochenen, Vektor-Befehle werden im Tracker über die OpenMP-Direktive `omp simd` [1] [2] verwendet. Diese Direktive allein weist den Compiler an, eine folgende Schleife ohne Verwendung von Threads zu parallelisieren, sondern pro Befehl von der CPU direkt mehrere aufeinanderfolgende Elemente gleichzeitig berechnen zu lassen. Die Anzahl der Iterationen einer solchen Schleife wird also entsprechend dividiert, durch die Zahl der von der Hardware bereitgestellten Einheiten. Die Direktive `omp simd` lässt sich, innerhalb

[1] SIMD: Single instruction multiple data, Klassifikation nach Flynn [Fly72]

[2] OpenMP, `omp simd` : `https://software.intel.com/en-us/node/693937`

eines mittels Threads parallelisierten Bereichs, auf eine zusätzliche Schleife anwenden. Die Direktive `omp parallel do simd` [1] dagegen würde eine Schleife auf Threads aufteilen, die dann wiederum jeweils ihren Anteil durch Vektorisierung abarbeiten. [HW10, Ope18]

Der Einsatz der Vektor-Befehle erfolgt im Tracker entweder auf einer weiteren inneren Schleifenebene oder es wird die innerste der drei Schleifen verwendet, wenn nur die äußeren beiden über `collapse(2)` parallelisiert sind.

Konkurrierender Zugriff auf geteilte Variablen innerhalb der parallelen Bereiche wird im Tracker bereits durch `omp atomic` gelöst, statt durch `omp critical`, so wie dies in Abschnitt 6.1.1 empfohlen worden ist. Hier besteht also kein Bedarf an Optimierung.

Allerdings führen die vielen in `omp parallel do` und `omp end parallel do` eingeschlossenen Abschnitte dazu, dass mehrfach Threads erzeugt und beendet werden, was jeweils etwas Zeit und Verwaltungsaufwand kostet. Hier wäre es eventuell sinnvoller, eine große parallele Region mit `omp parallel` und `omp end parallel` zu definieren und dazwischen lediglich mehrere Workshares über `omp do` und `omp end do` festzulegen. [arc18, HW10]

Die dazwischenliegenden sequentiellen Bereiche müsste man dann allerdings über `omp single` oder `omp master` auf nur einen Thread limitieren, ansonsten würden sie von jedem Thread identisch ausgeführt werden.

Während als Abschluss des sequentiellen Bereichs `omp end single` eine Barriere impliziert, ist dies bei `omp end master` nicht der Fall, so dass bei Bedarf noch ein `omp barrier` angehängt werden sollte. Welche Variante man wählt hängt vom Inhalt der sequentiellen Bereiche ab.

Wird im sequentiellen Bereich MPI verwendet, kann es sein, dass die MPI-Konfiguration nur MPI-Aufrufe aus Master-Threads zulässt. Dies wird von der MPI-Initialisierung bestimmt. Der Aufruf von `MPI_Init_thread()` [2] sieht folgende Varianten vor: [MPI18]

MPI_THREAD_SINGLE:
MPI-Methoden dürfen gar nicht aufgerufen werden, während das Programm in mehreren Threads arbeitet. Dies ist die Einstellung, die auch verwendet wird, wenn die Initialisierung über `MPI_Init()` statt `MPI_Init_thread()` erfolgt.

MPI_THREAD_FUNNELED:
Mehrere Threads sind zulässig, aber MPI-Aufrufe erfolgen nur im Master-Thread.

MPI_THREAD_SERIALIZED:
Aus allen Threads dürfen MPI-Aufrufe erfolgen, aber niemals mehr als einer zur Zeit.

MPI_THREAD_MULTIPLE:
MPI-Methoden dürfen ohne Einschränkungen verwendet werden.

Ist dagegen der Thread nicht festgelegt und der vorhergehende parallele Bereich tendenziell in der Last ungleich verteilt und mit einem `omp end do nowait` versehen, so kann `omp single`

[1] OpenMP, `omp parallel do simd` : `https://software.intel.com/en-us/node/693399`

[2] MPI_Init_thread: `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node303.htm`

die bessere Wahl sein. Beide Varianten sind allerdings nicht zulässig, wenn der Kontrollfluss darin z.B. durch ein `return` verlassen wird. Listing 6.2 zeigt einmal die Umsetzung dieses angepassten Schemas auf das, in Listing 6.1 gezeigte, einfache Beispiel.

Listing 6.1: Mehrmals Threads erzeugen

```
c$omp parallel do
      do i=0,N
       call shared_work(i)
      end do
c$omp end parallel do


      call single_work



      if(condition) then

       call conditional_work



c$omp parallel do
       do j=0,M
        call cond_shared_work(j)
       end do
c$omp end parallel do
      end if
```

Listing 6.2: Threads weiterverwenden

```
c$omp parallel
c$omp do
      do i=0,N
       call shared_work(i)
      end do
c$omp end do

c$omp master
      call single_work
c$omp end master
c$omp barrier

      if(condition) then
c$omp master
       call conditional_work
c$omp end master
c$omp barrier

c$omp do
       do j=0,M
        call cond_shared_work(j)
       end do
c$omp end do
      end if
c$omp end parallel
```

Als Nachteil dieses Schemas kann man auffassen, dass die `omp do`-Direktive zwar `private`, aber weder `default` noch `shared` Angaben zulässt. Man müsste also alle gemeinsam verwendeten Variablen in dem einleitenden `omp parallel` zusammenfassen oder sich auf die OpenMP Standardeinstellung `default(shared)` verlassen. Letzteres ist nicht ratsam, da es leicht zu Fehlern führen kann, wenn z.B. vergessen wurde eine Variable als privat zu deklarieren. Stattdessen wird bevorzugt `default(none)` anzugeben, was den Programmierer zwingt, jede verwendete Variable als geteilt oder privat zu benennen.

Letzterer Ansatz ist aber einmal testweise in PHOENIX umgesetzt worden[1]. Messungen des Speedups sind in Abbildung 6.2 dargestellt. Leider sind die Laufzeiten mit denen des Originals identisch. Trotz der Empfehlung, viele parallele Bereiche zu vermeiden, sind wohl

[1] Der Tracker im angepassten OpenMP-Schema ist auf der beigefügten CD in `Programmanpassungen/Source/3DRT/LC_Lstar_tracker_spherical_MultiPassCache.openmp.for` zu finden.

beide Compiler dazu in der Lage, entsprechende Optimierungen vorzunehmen. Eine manuelle Anpassung lohnt also in diesem Fall nicht.

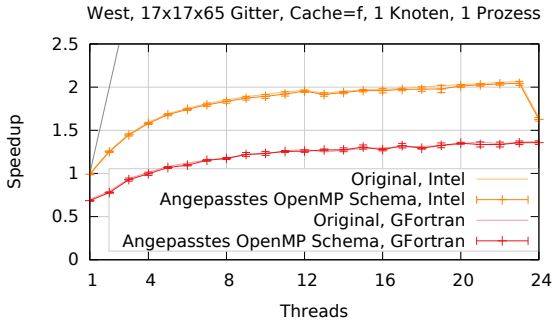

Abbildung 6.2.: Speedup nach Anpassung des OpenMP-Schemas
Trotz der empfohlenen Anpassungen am OpenMP-Schema liegt die Speedup-Kurve weiterhin auf der des Originals.

6.3. Variation des Gitteralgorithmus

Aus den vorherigen Überlegungen ergibt sich, dass die Reihenfolge, in der die Gitterelemente zum Erzeugen der Charakteristiken durchlaufen werden, entscheidend ist. Im Folgenden wird daher die, in Listing 6.3 gezeigte, dreifach verschachtelte Schleife über `it`, `ip` und `ir` angepasst und die Auswirkung auf die Leistung bestimmt.

6.3.1. Test der Asymmetrie

Zunächst soll einmal die Theorie bestätigt werden, dass die Reihenfolge der Schleifendurchläufe der Grund für die Asymmetrie der Raumwinkelabhängigkeit ist. Dazu wird eine der Schleifen angepasst und in ihrer Richtung umgekehrt. Da gezeigt wurde, dass die Abhängigkeit hauptsächlich auf dem Winkel θ beruht, wird der entsprechende Gitter-Index i_θ gewählt. Die Schleifenköpfe werden also angepasst wie in Listing 6.4 gezeigt.

Listing 6.3: Erzeugen der Charakteristiken
[Source/3DRT/LC_Lstar_tracker_spherical_MultiPassCache.for]

```
511       do it=-nt,nt
512        do ip=-np,np
513         do ir=-nr,nr
514 c--
515 c-- was this point hit already?
516 c--
517         if(chars_pv(ir,it,ip) .gt. 0) cycle
518 c--
519 c-- no, let's find a characteristic that goes through this point:
520         ...
```

Listing 6.4: Schleife mit negativer Schrittweite in i_θ
[Source/3DRT/LC_Lstar_tracker_spherical_MultiPassCache.for]

```
511      do it=nt,-nt,-1
512       do ip=-np,np
513        do ir=-nr,nr
514        ...
```

Nach dieser Anpassung und einer erneuten Datenaufnahme ergibt sich ein Verlauf wie in Abbildung 6.3 dargestellt. Der Verlauf der Abhängigkeit wurde nahezu in θ gespiegelt. Das zeigt, dass die Abhängigkeit in θ an die Reihenfolge gekoppelt ist, in der die Gitterelemente in i_θ durchlaufen werden. Das bestätigt also die Theorie, dass der Symmetriebruch durch die Reihenfolge des Gitterdurchlaufs verursacht wird. Eine analoge Kopplung wird zwischen ϕ und i_ϕ erwartet.

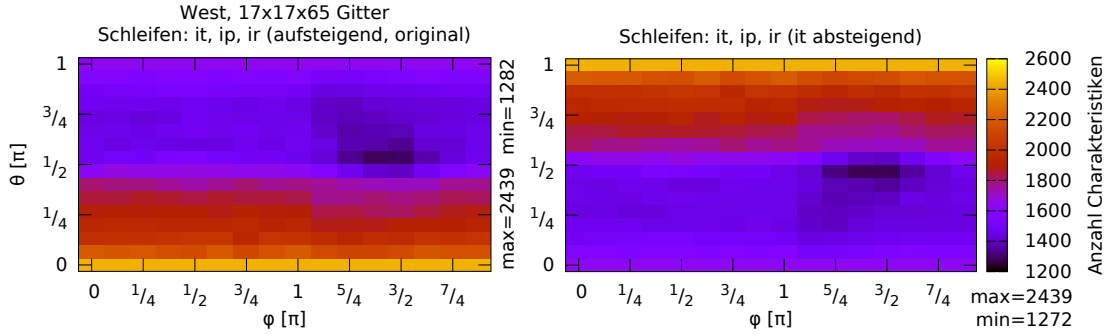

Abbildung 6.3.: Vergleich der Asymmetrie nach Umkehr der Schleife in i_θ
Links ist das Original und rechts das Ergebnis nach Umkehr der Schleifenrichtung in i_θ dargestellt. Die Asymmetrie in θ ist durch die Änderung ebenfalls umgekehrt worden. Das Ergebnis ist allerdings keine exakte Spiegelung, zu erkennen am geänderten Minimalwert, kommt dem aber sehr nahe.

6.3.2. Permutation der Schleifenindizes

Neben der zuvor gezeigten Änderung der Durchlaufrichtung einer Schleife, lässt sich auch die bestehende Anordnung von der äußeren zur inneren Schleife in Frage zu stellen, wie sie in Listing 6.3 gezeigt ist. Im Folgenden sollen alle 6 möglichen Permutationen der dreifach verschachtelten Schleife einmal betrachtet werden.

In der Auswertung hat sich gezeigt, dass sich die Verläufe sehr ähneln. Daher werden hier nicht alle entsprechenden Diagramme dargestellt. Es zeigt sich, dass es Diagramme vom "Typ A"

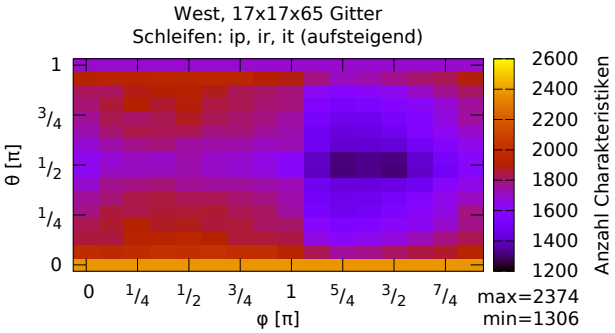

Abbildung 6.4.: Diagramm vom "Typ B"

gibt, die dem gezeigten Original gleichen, und einen weiteren "Typ B", der in Abbildung 6.4 gezeigt ist. Die aus allen Diagrammen entnommenen Kerndaten sind in Tabelle 6.2 zusammengefasst. Auf den Daten lassen sich folgende Beobachtungen machen:

- Diagramm Typ A tritt auf, wenn `it` außerhalb von `ip` iteriert wird, Typ B andernfalls.
- Iteriert die innerste Schleife über `ir`, ist die Spanne zwischen Minimum und Maximum am größten.
- Wird über `ir` in der äußersten Schleife iteriert, liefert dies den kleinsten Wert für das jeweilige Maximum.

Tabelle 6.2.: Ergebnisse der Schleifen-Permutationen

Schleifenindizes	Diagrammtyp	Minimum #Char.	Maximum #Char.
it, ip, ir (Original)	Typ A	1282	2439
it, ir, ip	Typ A	1304	2439
ir, it, ip	Typ A	1309	2374
ir, ip, it	Typ B	1330	2374
ip, ir, it	Typ B	1306	2374
ip, it, ir	Typ B	1297	2439

6.3.3. Leistungsvergleich angepasster Schleifen

Hier gilt als Ziel, das Maximum zu reduzieren, um damit auch die maximale Last der Prozesse zu verringern. Deshalb wird nun einmal die Leistung der Variante `ir, it, ip` gemessen, die dem Original noch ähnelt, aber das Maximum von 2439 Charakteristiken auf 2374 reduziert. Das Resultat in Abbildung 6.5 zeigt allerdings, dass diese kleine Änderung keinen Leistungsgewinn bringt. Die Leistung wird im Gegenteil sogar vermindert, was sich dadurch erklären lässt, dass das Zugriffsmuster der geänderten Schleife nun ungünstiger in Bezug auf die Ablage der Daten im Hauptspeicher ist.

Um den CPU-Cache geschickt zu nutzen, sollte das Zugriffsmuster der Schleifen zur Indizierung des Arrays passen. In Fortan werden Arrays spaltenweise im RAM abgelegt und auf das verwendete Array wird über die Indizes `ir, it, ip` zugegriffen. Spaltenweise bezieht sich auf die Darstellung einer Matrix als Array und bedeutet, der zuerst genannte Index wird sequentiell abgewickelt im RAM abgelegt und sollte daher zur effizienten Nutzung des CPU-Caches in der innersten Schleife durchlaufen werden. Demzufolge sollte die Cache-optimierte Schleifenreihenfolge `ip, it, ir` lauten. [Ben18]

Zur Analyse sind einmal mit `perf` die dort zugänglichen Cache-Statistiken ermittelt worden[1]. Tabelle 6.3 zeigt die Ergebnisse. Diese lassen sich zumindest so auslegen, dass die vorigen Annahmen bestätigt werden, auch wenn die Deutung teilweise schwer fällt.

Auch die Cache-optimierte Variante ist einmal in Abbildung 6.5 dargestellt und zeigt tatsächlich eine Beschleunigung, zumindest bei Verwendung des Intel Compilers. Nun stellt sich

[1] perf Tutorial: `https://perf.wiki.kernel.org/index.php/Tutorial`

die Frage, warum in PHOENIX die Original Schleifenreihenfolge `it, ip, ir` lautet und ob die Cache-Optimierung immer die beste Wahl ist. Kann das Maximum weiter reduziert werden als in der gezeigten Variante, sollte dies irgendwann einen größeren Effekt erzielen als die Cache-Zugriffe. Im Folgenden soll daher gezeigt werden, wie das Maximum weiter reduziert werden kann.

Tabelle 6.3.: Cache-Effizienz im Vergleich

	L1		L3		Gesamt	
Schleifen-Variante	Loads	Misses	Loads	Misses	References	Misses
Original	180 G	1,90 %	1,64 G	5,58 %	2,17 G	8.62 %
Maximum reduziert	186 G	2,01 %	1,84 G	5,17 %	2,39 G	8,60 %
Cache-optimiert	169 G	1,94 %	1,59 G	5,07 %	2,05 G	8.19 %

Die Zahlen der Cache-Zugriffe wurden bei einem Prozess auf kleinem Gitter in der Intel-Variante mit `perf` erhoben. Zumindest die Rate der *L1 Misses* (bei etwa gleicher Gesamtrate) deutet an, dass bei der Variante mit leichter Reduktion des Maximums, im Vergleich zum Original, leider der Cache etwas schlechter genutzt wird. Die theoretisch für den Cache optimierte Variante zeigt hier zumindest bei der Gesamtrate Wirkung.

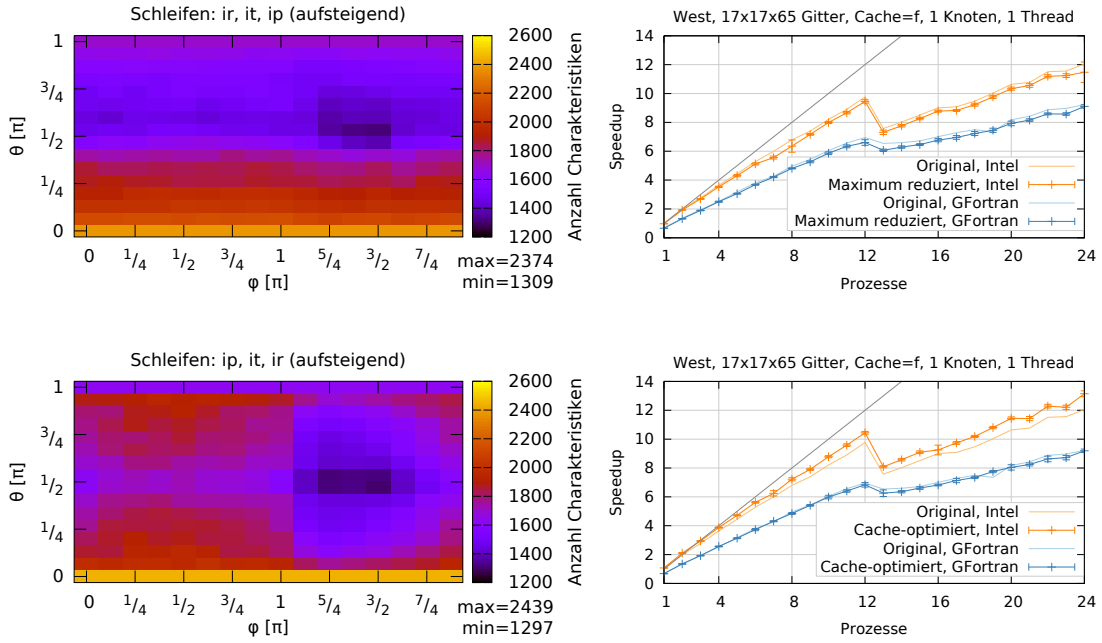

Abbildung 6.5.: Vergleich der Leistung nach Anpassung der Schleifen

Die angepasste Schleife erzeugt einen Verlauf der Raumwinkelabhängigkeit wie links dargestellt. Der dazu gemessene Speedup wird rechts daneben angezeigt und mit dem Original verglichen. Die obere leichte Reduktion des Maximums zeigt leider eine Minderung der Leistung (Laufzeit von 12 Prozessen bei Intel auf 103,4%, bei GFortran auf 105,3% gestiegen). Unten wird einmal die Variante gezeigt, in der die Schleifenreihenfolge zur Ablage des Arrays im Speicher passt und daher für den CPU-Cache optimiert ist, was die Leistung verbessern kann (Laufzeit von 12 Prozessen bei Intel auf 93,6% reduziert, bei GFortran auf 101,8% gestiegen).

6.3.4. Implementation einer Raumwinkelabhängigkeit

Aus der Erkenntnis, dass die Last vom Raumwinkel abhängt und sich dies durch Anpassung der gezeigten Schleife beeinflussen lässt, ergibt sich folgender Ansatz: Es sollte ein adaptiver Algorithmus verwendet werden, der den gegebenen Raumwinkel (θ, ϕ) berücksichtigt.

In Abbildung 6.3 ist gezeigt worden, wie der Verlauf der θ-Abhängigkeit umgekehrt wird, indem i_θ bzw. `it` mit der Schrittweite -1 durchlaufen wird. Die Diagramme zeigen eine wünschenswerte Eigenschaft, denn im Original wird für $\theta \geq \frac{\pi}{2}$ eine geringe Anzahl Charakteristiken benötigt, wohingegen bei Schrittweite -1 für $\theta \leq \frac{\pi}{2}$ die Zahl gering ist. Wird also abhängig von θ zwischen den Schrittweiten 1 und -1 gewählt, sollte sich aus beiden Verläufen nur die wünschenswerte Hälfte selektieren lassen.

Die Schleife wird nun also entsprechend, wie in Listing 6.5 gezeigt, angepasst. Die resultierende Raumwinkelabhängigkeit und die Ergebnisse der Leistungsmessung sind in Abbildung 6.6 gezeigt. Der gewünschte Effekt ist erreicht worden, das Maximum der benötigten Charakteristiken wurde von 2439 auf 1628 deutlich reduziert. Das wiederum verringert die Last soweit, dass der Speedup eine Leistungssteigerung bei beiden Compilern anzeigt.

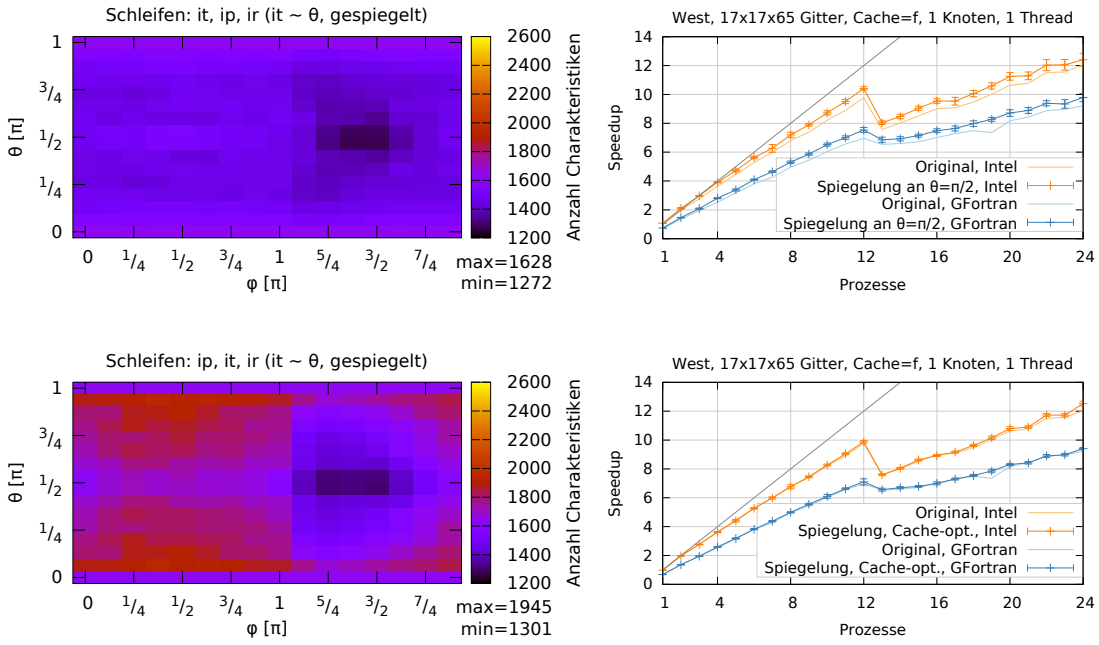

Abbildung 6.6.: Vergleich der Leistung nach Implementation einer Raumwinkelabhängigkeit
Die Schleifenrichtung in `it` ist nun abhängig von θ, in der Art, dass sich der oben links gezeigte wünschenswerte Verlauf, bei Erhaltung der Originalreihenfolge `it, ip, ir`, ergibt. Das Maximum ist von 2439 auf 1628 reduziert. Dadurch zeigt der Speedup oben rechts nun eine Verbesserung der Leistung (Laufzeit von 12 Prozessen bei Intel auf 93,5%, bei GFortran auf 93,1% reduziert). Die unten dargestellte Cache-optimierte Schleifenreihenfolge in Kombination mit der θ-Abhängigkeit, kann das Maximum nicht so stark reduzieren und erzielt daher auch keinen nennenswerten Speedup.

Listing 6.5: Schleife mit Schrittweite abhängig vom Raumwinkel
[Source/3DRT/LC_Lstar_tracker_spherical_MultiPassCache.for]

```
46      integer :: itstart,itend,itstep
47      ...
512     if(theta .lt. pi/2) then
513      itstart = nt
514      itend = -nt
515      itstep = -1
516     else
517      itstart = -nt
518      itend = nt
519      itstep = 1
520     endif
521     do it=itstart,itend,itstep
522      do ip=-np,np
523       do ir=-nr,nr
524        ...
```

Vergleicht man das Ergebnis nun wieder mit der Cache-optimierten Schleifenreihenfolge `ip, it, ir`, wobei aber weiterhin `it` wie beschrieben von θ abhängt, so zeigt sich, dass dies das Maximum nicht so stark reduziert und die Leistung im Vergleich zum Original kaum verbessert. Dies betont, dass in diesem Fall die Reduktion des Maximums entscheidend ist, selbst wenn die dabei verwendete Schleifenreihenfolge nicht für den CPU-Cache optimiert ist.

Erstellt man analog zu Abschnitt 4.2 ein Profil der verbesserten Lösung (Schleifenreihenfolge `it, ip, ir`), so kann man in den Ergebnissen in Abbildung 6.7 sehen, wenn man sie mit Abbildung 4.7 des Originals vergleicht, dass der Anteil der `MPI_Barrier` von 3,6% auf 2,7% gesunken ist. Damit ist gezeigt, dass das Ziel, die ungleiche Lastverteilung etwas auszugleichen, erreicht worden ist und dies, neben der insgesamt von der Anzahl Charakteristiken abhängenden Last, zu der gezeigten Leistungsverbesserung führt.

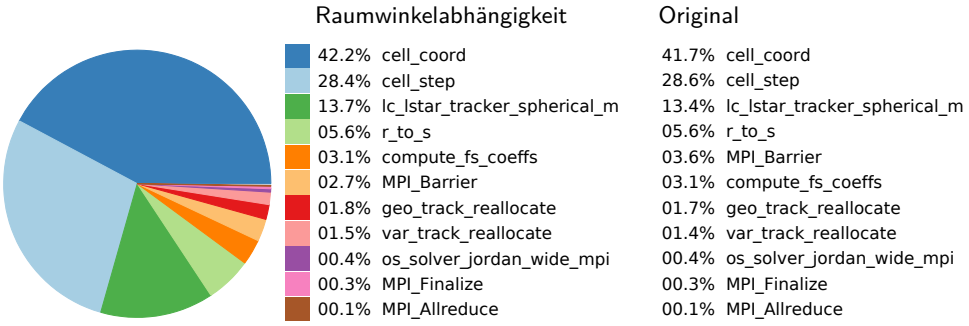

Abbildung 6.7.: Profil nach Implementation einer Raumwinkelabhängigkeit
Hier sind die Ergebnisse des Profilings nach Implementation der Raumwinkelabhängigkeit zusammengefasst. Im Vergleich zum Original (rechts, Wdh. der Daten aus Abbildung 4.7) ist der Anteil der `MPI_Barrier` um einen Platz von 3,6% auf 2,7% gesunken.

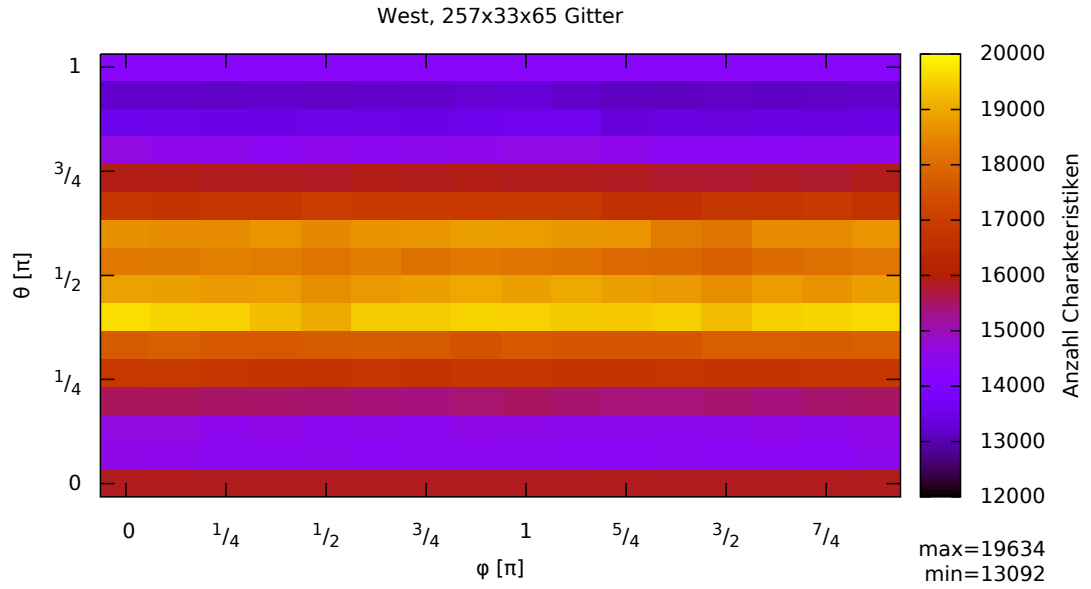

Abbildung 6.8.: Raumwinkelabhängigkeit auf großem Gitter
Die Anzahl Charakteristiken in Abhängigkeit zum Raumwinkel (θ, ϕ) ist hier für das große Gitter dargestellt. Die Spanne zwischen Maximum und Minimum ist auch hier sehr groß, ebenso wird der Verlauf durch θ dominiert. Im Gegensatz zum kleinen Gitter ist das Maximum hier aber nahe $\theta \approx \frac{\pi}{2}$ angesiedelt statt bei $\theta = 0$.

6.3.5. Validierung bisheriger Ergebnisse auf großem Gitter

Die im vorigen Abschnitt erreichte Leistungssteigerung basiert auf Beobachtungen an einem konkreten Fall, dem gewählten kleinen 17×17×65 Gitter. In jedem Fall wurde gezeigt, dass der Ansatz, die Erzeugung der Charakteristiken vom betrachteten Raumwinkel (θ, ϕ) abhängig zu machen, positive Ergebnisse erzielen kann. Wie genau die Anzahl der Charakteristiken vom Raumwinkel abhängt und welche Variation der Schleifen die Leistung steigert, ist für ein beliebiges Gitter aber weiterhin unklar. Ein allgemeingültiger Zusammenhang wird hier nicht hergestellt werden können, aber es soll dennoch einmal das große 257×33×65 Gitter betrachtet werden.

Mit analogen Messungen, wie sie im Abschnitt 4.4.1 beschrieben worden sind, ergibt sich ein Verlauf der Anzahl Charakteristiken, in Abhängigkeit zum Raumwinkel, wie in Abbildung 6.8 dargestellt. Obwohl das 257×33×65 Gitter etwa 30 mal so viele Gitterelemente enthält wie das kleine 17×17×65 Gitter, liegt die Anzahl der benötigten Charakteristiken um etwa Faktor 10 höher. Dies liegt offensichtlich daran, dass das Gitter durchquerende Charakteristiken jeweils deutlich mehr Elemente treffen. Die Spanne zwischen der maximalen und minimalen Anzahl benötigter Charakteristiken ist beinahe proportional gewachsen, das Maximum liegt nun bei dem etwa 1,5-fachen des Minimums (vgl. Faktor 1,9 bei kleinem Gitter).

Weiterhin bestimmt im Wesentlichen θ die Anzahl der Charakteristiken, ϕ spielt eine untergeordnete Rolle. Im Gegensatz zum deutlichen Gefälle auf dem kleinen Gitter, vom Maximum

bei $\theta = 0$ zu kleineren Anzahlen der Charakteristiken bei wachsendem θ, liegt auf dem großen Gitter das Maximum nahe $\theta \approx \frac{\pi}{2}$, umgeben von einem Gürtel hoher Werte und einem Gefälle der Anzahl Charakteristiken sowohl in wachsenden wie fallenden θ. Daraus ergibt sich direkt, dass eine Spiegelung der oberen Hemisphäre $\theta > \frac{\pi}{2}$ auf die untere nicht mehr alle hohen Werte ausblenden kann, wie dies auch in Abbildung 6.9 oben dargestellt ist.

Auch wenn die Reduktion des Maximums, durch das in Abschnitt 6.3.4 beschriebene Verfahren, relativ betrachtet nur klein ist (was auf dem kleinen Gitter noch keinen Leistungsgewinn brachte), so ist es absolut betrachtet eine beachtliche Anzahl Charakteristiken, die nicht vom Tracker berechnet werden muss. Da auch der Aufwand für jede einzelne Charakteristik steigt (sie muss durch ein größeres Gitter verfolgt werden), führt die Reduktion der maximalen Anzahl Charakteristiken auch hier zu einem Leistungsgewinn, gemessen am erreichten Speedup, wie rechts oben in Abbildung 6.9 zu sehen ist.

Auf dem großen Gitter gilt analog zum kleinen, wie unten in Abbildung 6.9 zu sehen, dass die zur Ablage im RAM passende Schleifenreihenfolge zwar den CPU-Cache nutzt, aber ir-

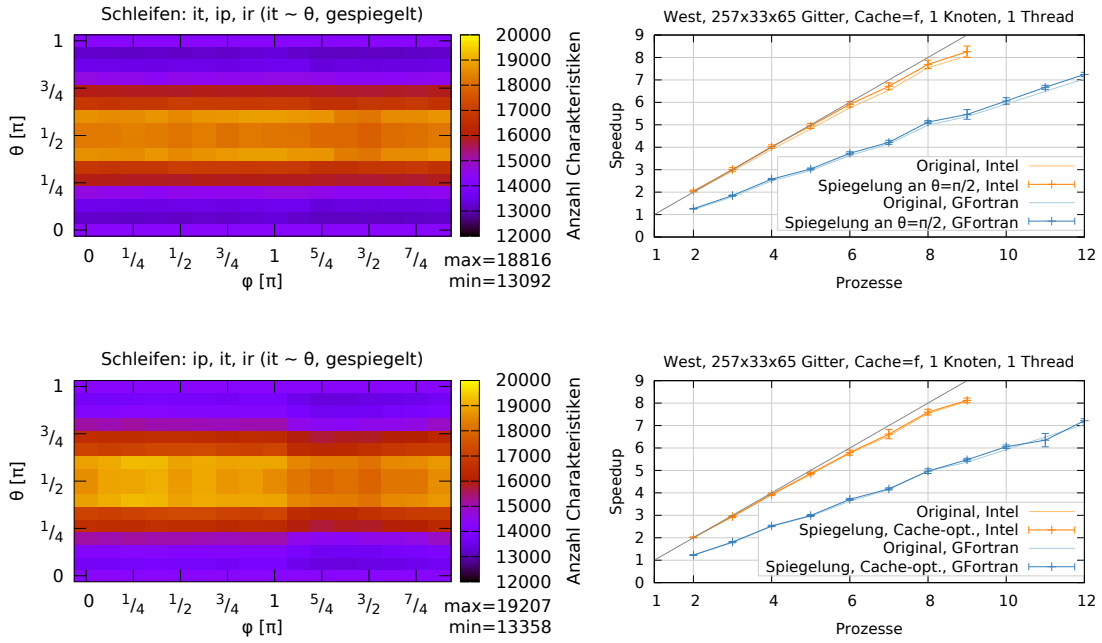

Abbildung 6.9.: Angepasste Schleifen in Abhängigkeit zu θ auf großem Gitter

Gezeigt ist die Anwendung des Algorithmus, der abhängig von θ zwischen der Schrittweite 1 und -1 in `it` wählt, auf das große Gitter. Oben ist die Schleifenreihenfolge `it, ip, ir` dargestellt, die auf dem kleinen Gitter die Maximalzahl der benötigten Charakteristiken deutlich reduzieren konnte. Hier fällt die Reduktion der maximalen Anzahl Charakteristiken von 19634 auf 18816 zwar relativ gesehen geringer aus, in absoluten Zahlen ist sie aber dennoch beachtlich, was zu einer Leistungsteigerung, gemessen am Speedup rechts, führt (Laufzeit von 8 Prozessen bei Intel auf 97,3%, bei GFortran auf 97,2% reduziert). Unten ist im Vergleich dazu die Schleifenreihenfolge gezeigt, die auf effiziente Nutzung des CPU-Caches setzt. Auch hier kann die obere Variante wieder einen größeren Leistungsgewinn durch stärkere Reduktion des Maximums erzielen.

relevant wird, sobald eine andere Schleifenreihenfolge eine Reduktion der Charakteristiken erreicht, was stärker zur Leistungsverbesserung beiträgt.

Insgesamt gilt also auch auf dem großen Gitter, dass die von θ abhängige Wahl der Schrittweite 1 oder -1 in i_θ die Leistung steigert. Dies stellt nur eine recht elementare Variation des originalen Algorithmus dar. Zugrunde gelegt wurde die Beobachtung, dass es einen Zusammenhang zwischen dem Raumwinkel θ und dem Gitterindex i_θ gibt, derart dass eine Spiegelung des Raumwinkels an $\theta = \frac{\pi}{2}$ durch eine Umkehr der Schrittrichtung kompensiert wird. Dies soll als allgemeingültig angenommen werden. Nimmt man weiterhin an, dass eine positive Schrittweite stets zur Lage des Maximums bei $0 \leq \theta \leq \frac{\pi}{2}$ führt, so wird das beschriebene Verfahren die Anzahl der Charakteristiken niemals erhöhen, sondern stets senken oder keine Auswirkung zeigen. Entsprechend wird auch die Leistung von PHOENIX durch dieses Verfahren i.d.R. positiv beeinflusst.

6.3.6. Verzögerte Betrachtung des innersten Radius

Neben der Variation der Reihenfolge der drei verschachtelten Schleifen, sowie der Änderung der Laufrichtung einzelner Schleifenindizes, lässt sich auch noch ein weiterer Ansatz verfolgen. Ein Schleifenindex muss nicht beim minimalen oder maximalem Wert beginnen. Ein Index kann an einer beliebigen Position (Offset) innerhalb des Intervalls starten, dann in eine Richtung bis zum Ende laufen, um dann vom anderen Ende beginnend die verbleibenden Elemente abzuarbeiten. Auch solch eine Variation im Algorithmus zur Erzeugung der Charakteristiken, wird das Ergebnis, die Anzahl der Charakteristiken, in irgendeiner Form beeinflussen und soll daher nun einmal betrachtet werden.

In Abschnitt 4.4.3 ist gezeigt worden, dass das sphärische Gitter in der Schicht des innersten Radius eine auffällig hohe Dichte an Charakteristiken aufweist. Da dies eine Besonderheit bei einem einzelnen Index i_r darstellt, soll auf den Schleifendurchlauf der Radien einmal der Offset 1 angewendet werden, also die innerste Schicht zunächst übersprungen und in der zweiten Schicht von innen begonnen werden. Nachdem dann von innen nach außen alle

Listing 6.6: Schleife mit Offset im Index i_r
[Source/3DRT/LC_Lstar_tracker_spherical_MultiPassCache.for]

```
46      integer :: tmpir
47      ...
512     do it=-nt,nt
513      do ip=-np,np
514       do tmpir=-nr,nr
515        if(tmpir .eq. nr) then
516         ir = -nr
517        else
518         ir = tmpir + 1
519        endif
520        ...
```

Schichten durchlaufen sind, wird die innerste betrachtet. Die Umsetzung ist in Listing 6.6 gezeigt.

Der Verlauf der Anzahl Charakteristiken, in Abhängigkeit vom Raumwinkel, bei Anwendung dieses Offsets, ist in Abbildung 6.10 dargestellt. Auch auf dem großen Gitter ergibt sich ein Verlauf, der dem Original ähnelt und sich hauptsächlich im erreichten Minimum und Maximum unterscheidet. Die Ergebnisse sind in Tabelle 6.4 zusammengefasst.

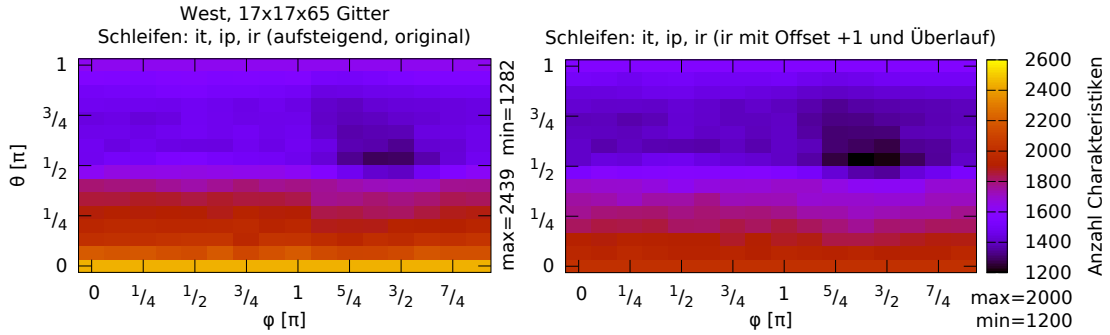

Abbildung 6.10.: Verlauf der Anzahl Charakteristiken durch Offset 1 in i_r

Links ist das Original und rechts das Ergebnis mit Offset 1 in der i_r Schleife dargestellt. Die innerste Schicht in i_r auszulassen (Offset 1) bewirkt, wie zu sehen ist, eine Reduktion der Anzahl Charakteristiken.

Tabelle 6.4.: Ergebnisse des Offsets 1 in i_r

Gitter	Schleifen-Variante	Minimum #Char.	Maximum #Char.
17×17×65	Original	1282	2439
17×17×65	Offset 1 in i_r	1200	2000
17×17×65	Schrittweite 1 oder -1 in i_θ	1272	1628
17×17×65	Schrittweite + Offset 1	1195	1562
257×33×65	Original	13092	19634
257×33×65	Offset 1 in i_r	12999	19297
257×33×65	Schrittweite 1 oder -1 in i_θ	13092	18816
257×33×65	Schrittweite + Offset 1	12999	18547

Es werden die Offset-Varianten mit dem Original und der Schrittweite 1 oder -1 in Abhängigkeit vom Raumwinkel θ verglichen. Auch die Kombination von Offset und Schrittweite ist dargestellt und erzielt die niedrigste Anzahl benötigter Charakteristiken.

Da die Messungen mit Implementation des Offsets die Anzahl der benötigten Charakteristiken reduzieren, ist in Tabelle 6.4 auch der Vergleich mit der Variation der Schrittweite gezeigt. Ebenso ist auch die Kombination beider Eingriffe implementiert und vermessen worden. Listing 6.7 zeigt die Implementierung. Diese erzielt, wie in der Tabelle gezeigt, die bisher größte Reduktion der Charakteristiken. Daraus resultiert dann wieder eine Leistungssteigerung, wie sie in Abbildung 6.11 gezeigt ist.

Listing 6.7: Kombination der Schrittweite und des Offsets
[Source/3DRT/LC_Lstar_tracker_spherical_MultiPassCache.for]

```
46      integer :: itstart,itend,itstep,tmpir
47      ...
512     if(theta .lt. pi/2) then
513      itstart = nt
514      itend = -nt
515      itstep = -1
516     else
517      itstart = -nt
518      itend = nt
519      itstep = 1
520     endif
521     do it=itstart,itend,itstep
522      do ip=-np,np
523       do tmpir=-nr,nr
524        if(tmpir .eq. nr) then
525         ir = -nr
526        else
527         ir = tmpir + 1
528        endif
529        ...
```

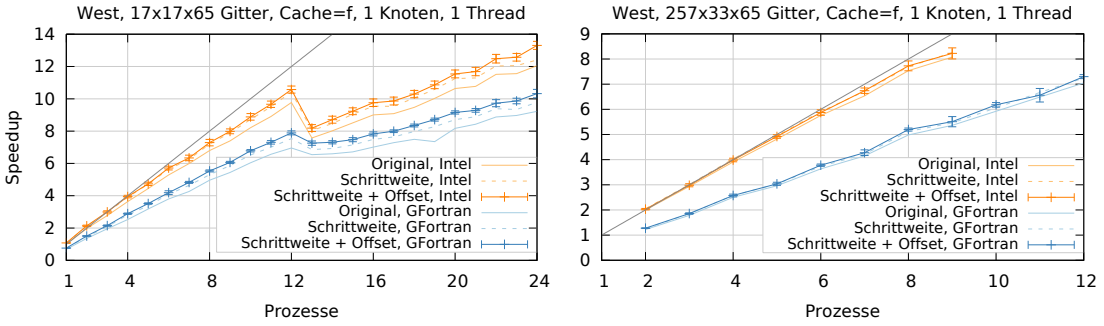

Abbildung 6.11.: Leistungssteigerung durch Kombination von Schrittweite und Offset
Links ist die Messung mit dem kleinen und rechts mit dem großen Gitter zu sehen. Verwendet wird die Schleifenvariante mit Schrittweite 1 oder -1 im Index i_θ, abhängig vom Raumwinkel θ, in Kombination mit dem Offset 1 auf dem Index i_r, also einer verzögerten Betrachtung der innersten Schicht des Gitters. Im Vergleich zum Original ist auch hier die Leistung verbessert (Laufzeit von 12 Prozessen auf kleinem Gitter bei Intel auf 92,4%, bei GFortran auf 88,1% reduziert; von 8 Prozessen auf großem Gitter bei Intel auf 97,5%, bei GFortran auf 95,8%).

In der Beschreibung des sphärischen Trackers in Abschnitt 2.3 wurde allerdings erwähnt, dass der Aufbau der Schichten, speziell die Abbildung des Index i_r auf einen Radius R, frei über eine Zuordnungstabelle `Rmap` definierbar ist. Die Wahl der Radien wird die Dichte der Charakteristiken pro Gitterelement beeinflussen, so dass nicht immer gegeben sein wird, dass die innerste Schicht, wie zuvor beschrieben, auffällt. Die Methode des Offsets scheint daher weniger allgemein verwendbar als die Spiegelung / Schrittweitenvariation. Sofern eine konkrete `Rmap` vorliegt, ist eine entsprechende Analyse anzuraten.

7. Fazit

Auf einer umfangreichen Leistungsanalyse und theoretischen Überlegungen basierend, sind mehrere Ansätze zur Verbesserung der Leistung von PHOENIX verfolgt worden.

Obwohl die Skalierung des Speedups beim Einsatz von OpenMP als verbesserungswürdig aufgezeigt wurde, ist hier kein Erfolg erreicht worden. Gängige Empfehlungen, mehrfache parallele Regionen in OpenMP zu vermeiden, und stattdessen die Definition paralleler Bereiche von der Arbeitsteilung auf Schleifenkonstrukten zu trennen, zeigen im Experiment keinerlei Auswirkung auf die Leistung. Es wird vermutet, dass solche Eingriffe durch die Optimierungen der Compiler heute unnötig geworden sind.

Als Maß für die Last, die im Programmlauf innerhalb eines Aufrufs des Trackers zu bewältigen ist, ist die Anzahl der Charakteristiken identifiziert worden, die das sphärische Gitter durchlaufen. Ansätze, diese Anzahl zu reduzieren, sind erfolgreich umgesetzt worden, was auch die Laufzeit von PHOENIX entsprechend verkürzt:

Raumwinkelabhängigkeit (Schrittweite 1 oder -1 in i_θ)
: Es ist gezeigt worden, dass die Last vom Raumwinkel (θ, ϕ) der Charakteristiken abhängt. Wird bei der Methode zur Erzeugung der Charakteristiken, der gerade betrachtete Raumwinkel berücksichtigt, kann diese Abhängigkeit zum positiven beeinflusst werden. Allein die Wahl der Schrittweite 1 oder -1 einer Schleife, jeweils für eine Teilmenge der θ, zeigt schon einen nennenswerten Leistungsgewinn. Die bei dieser elementaren Implementation zugrunde gelegten Einsichten werden als allgemeingültig verstanden. Es wird daher angenommen, dass diese Methode für beliebig definierte sphärische Gitter ebenfalls die Leistung steigert.

Verzögerte Betrachtung des innersten Radius (Offset 1 in i_r)
: In der Betrachtung eines Beispiels wurde erkannt, dass die Gitterelemente des innersten Radius eine besonders hohe Dichte an Charakteristiken aufwiesen, die sie durchlaufen. Die Idee, nicht mit dieser Schicht bei der Erzeugung der Charakteristiken zu beginnen, sondern die entsprechende Schleife mit einem Offset von 1 zu durchlaufen, hat ebenfalls zu einer Reduktion der Charakteristiken geführt. Dieser Ansatz lässt sich mit dem erstgenannten kombinieren, gilt jedoch als weniger allgemeingültig und ist von der Festlegung einer Gitter-Geometrie, speziell der Wahl der Radien, abhängig.

Da in PHOENIX bereits diverse Ansätze, zur Parallelisierung und Leistungsverbesserung im Allgemeinen, implementiert waren, sind in Folgearbeiten nur sukzessive kleinere Leistungsgewinne zu erwarten. Eine Reduktion der Laufzeit um wenige Prozent wird daher schon als Fortschritt betrachtet.

Die Programmlaufzeiten bei Einsatz der genannten Methoden zur Leistungsverbesserung sind in Abbildung 7.1 zusammengefasst. Die Implementation der Raumwinkelabhängigkeit durch variable Schrittweite liefert, bei Verwendung beider Compiler, ein gutes Ergebnis und wird daher zur generellen Übernahme in PHOENIX empfohlen. Die Kombination beider Methoden kann die Leistung zusätzlich steigern, sofern die Offset-Methode bei der Wahl der Radien des Gitters sinnvoll erscheint.

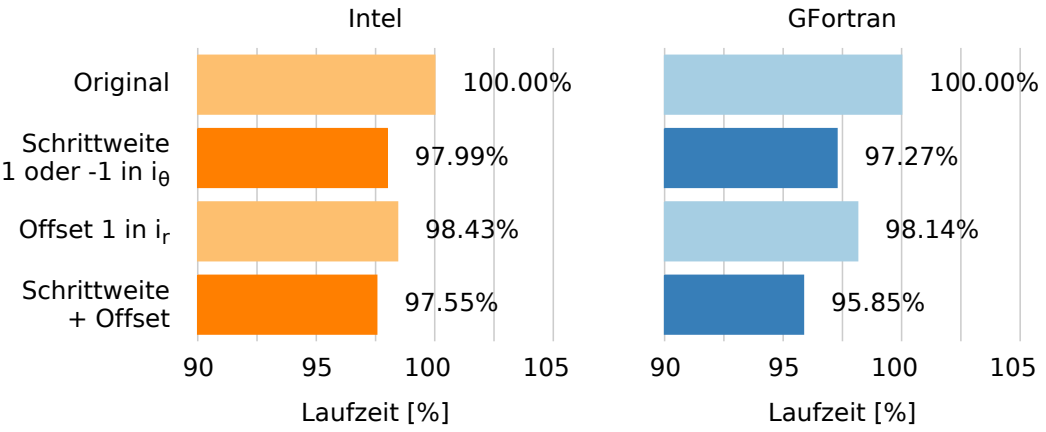

Abbildung 7.1.: Relative Laufzeiten der verbesserten Varianten

Es sind die Laufzeiten der Varianten, die in Kapitel 6 als Verbesserungen ermittelt wurden, in Relation zum Original dargestellt. Verglichen werden Programmläufe mit 8 Prozessen auf dem großen 257×33×65 Gitter. Die 100% entsprechen einer Laufzeit von 58 Minuten bei Intel und 88 Minuten bei GFortran.

8. Empfohlene weitere Maßnahmen

Die in der Leistungsanalyse, speziell im Profiling, hervorgehobenen Funktionen, die häufig aufgerufen werden und große Anteile der Laufzeit ausmachen, konnten in dieser Arbeit nicht mittels OpenMP angegangen werden. Verbesserungen dieser Funktionen würden aber sicherlich einen nennenswerten Leistungsgewinn erzielen. Hier sollte anhand der zugrundeliegenden Mathematik geprüft werden, ob sich nicht äquivalente Darstellungen finden lassen, die in einem effizienteren Algorithmus umgesetzt werden können.

Insbesondere besteht ein Großteil der Berechnungen aus Koordinatentransformationen, von kartesisch zu sphärisch, und es könnte evtl. auch sinnvoll sein, die Charakteristiken durchgehend in sphärischen Koordinaten zu betrachten, wenngleich sich ihre geradlinige Entwicklung (kartesisch) dadurch deutlich schwieriger darstellen lässt.

Als technisch begrenzender Faktor, ergibt sich die maximal einsetzbare Zahl an Prozessen, wie beschrieben, aus der Auflösung der Wellenlängen und Raumwinkel. Dies ist durch die Ebenen der MPI-Parallelisierung festgelegt. Mehr MPI-Prozesse ausschließlich in die Bearbeitung immer größerer Gitter zu investieren ist aktuell nicht vorgesehen. Um dies zu realisieren, könnte eine weitere MPI-Ebene eingeführt werden, so dass mehrere Prozesse an dem gleichen Raumwinkel arbeiten können. Dies würde vermutlich neue Herausforderungen, wie zusätzlichen Kommunikationsaufwand, mit sich bringen, aber ermöglichen, flexibler auf unterschiedliche Anforderungen der Anwender zu reagieren.

Für jeden konkreten Anwendungsfall wird ebenfalls empfohlen, die Möglichkeiten des jeweils aktuellen Compilers im Zusammenspiel mit den auf dem Zielsystem eingesetzten Bibliotheken zu evaluieren und über mögliche Compiler-Optionen das Maximum an Leistung zu erreichen.

Die Abhängigkeit der benötigten Anzahl Charakteristiken vom Raumwinkel ist ein interessanter Aspekt, der den Rechenaufwand beeinflusst und in dieser Arbeit nur in Grundzügen aufgezeigt wurde. Eine genauere Analyse dieses Problems, mit dem Ziel den Verlauf und die Lage der Extrema für beliebig vorgegebene Gitter vorherzusagen, würde sicherlich weitere Möglichkeiten zur Verbesserung der Leistung bieten und wäre wohl auch in anderen Anwendungen einsetzbar.

Die vorgestellte einfache Methode der Implementation einer Raumwinkelabhängigkeit zur Erzeugung der Charakteristiken ließe sich vermutlich auch auf weitere Tracker übertragen, die ein sphärisches Gitter verwenden (die Varianten ohne MultiPass und Cache). Diese könnten dahingehend überprüft werden.

Ob sich ein vergleichbares Verhalten auch auf Zylinderkoordinaten zeigt und sich mit analogen Ansätzen angehen lässt, könnte ebenso Gegenstand weiterer Betrachtungen sein.

Literaturverzeichnis

[arc18] ARCHER: *Advanced OpenMP: Tips, Tricks & Gotchas*. https://www.archer.ac.uk/training/course-material/2016/08/160802_AdvOpenMP_Bristol. Version: 07 2018

[Ben18] BENDERSKY, Eli: *Memory layout of multi-dimensional arrays*. https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays. Version: 07 2018

[Ble17] BLESEL, Michael: *Compiler assisted translation of OpenMP to MPI using LLVM*, Universität Hamburg, Bachelor's Thesis, 10 2017

[BM06] BAUKE, H. ; MERTENS, S.: *Cluster Computing: Praktische Einführung in das Hochleistungsrechnen auf Linux-Clustern*. Springer, 2006 https://www.springer.com/de/book/9783540422990. – ISBN 978–3–540–29928–8

[CDL+17] CHAUVIN, G. ; DESIDERA, S. ; LAGRANGE, A.-M. ; VIGAN, A. ; GRATTON, R. ; LANGLOIS, M. ; BONNEFOY, M. ; BEUZIT, J.-L. u. a.: Discovery of a warm, dusty giant planet around HIP 65426. In: *Astronomy and Astrophysics* 605 (2017), September, S. L9. http://dx.doi.org/10.1051/0004-6361/201731152. – DOI 10.1051/0004–6361/201731152

[CLD+04] CHAUVIN, G. ; LAGRANGE, A.-M. ; DUMAS, C. ; ZUCKERMAN, B. ; MOUILLET, D. ; SONG, I. ; BEUZIT, J.-L. ; LOWRANCE, P.: A giant planet candidate near a young brown dwarf. Direct VLT/NACO observations using IR wavefront sensing. In: *Astronomy and Astrophysics* 425 (2004), Oktober, S. L29–L32. http://dx.doi.org/10.1051/0004-6361:200400056. – DOI 10.1051/0004–6361:200400056

[Fly72] FLYNN, Michael J.: Some Computer Organizations and Their Effectiveness. In: *IEEE Transactions on Computers* C-21 (1972), S. 948–960

[FP10] FRIEDRICH, H. ; PIETSCHMANN, F.: *Numerische Methoden: ein Lehr- und Übungsbuch*. De Gruyter, 2010 (De Gruyter Studium). https://books.google.de/books?id=do9YDTsiPggC. – ISBN 978–3–11–021806–0

[Ham18] HAMBURGER STERNWARTE: *PHOENIX*. http://www.hs.uni-hamburg.de/EN/For/ThA/phoenix/index.html. Version: 07 2018

[HB06] HAUSCHILDT, P. H. ; BARON, E.: A 3D radiative transfer framework. I. Non-local operator splitting and continuum scattering problems. In: *Astronomy and*

Astrophysics 451 (2006), Mai, S. 273–284. http://dx.doi.org/10.1051/0004-6361:20053846. – DOI 10.1051/0004–6361:20053846

[HB09] HAUSCHILDT, P. H. ; BARON, E.: A 3D radiative transfer framework. IV. Spherical and cylindrical coordinate systems. In: *Astronomy and Astrophysics* 498 (2009), Mai, S. 981–985. http://dx.doi.org/10.1051/0004-6361/200911661. – DOI 10.1051/0004–6361/200911661

[HB11] HAUSCHILDT, P. H. ; BARON, E.: A 3D radiative transfer framework. VIII. OpenCL implementation. In: *Astronomy and Astrophysics* 533 (2011), September, S. A127. http://dx.doi.org/10.1051/0004-6361/201117051. – DOI 10.1051/0004–6361/201117051

[HW10] HAGER, G. ; WELLEIN, G.: *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010 (Chapman & Hall/CRC Computational Science). http://www.crcpress.com/product/isbn/9781439811924. – ISBN 978–1–4398–1192–4

[LLV18] LLVM: *Writing an LLVM Pass*. http://llvm.org/docs/WritingAnLLVMPass.html. Version: 07 2018

[LSB18] LACY, B. ; SHLIVKO, D. ; BURROWS, A.: Characterization of Exoplanet Atmospheres with the Optical Coronagraph on WFIRST. In: *ArXiv e-prints* (2018), Januar

[Lud18] LUDWIG, Thomas: *Hochleistungsrechnen*. Vorlesung, Universität Hamburg. https://wr.informatik.uni-hamburg.de/teaching/wintersemester_2017_2018/hochleistungsrechnen. Version: 2017/18

[MPI18] MPI FORUM: *Message Passing Interface (MPI)*. https://www.mpi-forum.org. Version: 07 2018

[NAS18a] NASA: *Exoplanet Archive*. https://exoplanetarchive.ipac.caltech.edu. Version: 07 2018

[NAS18b] NASA: *Kepler and K2*. https://www.nasa.gov/mission_pages/kepler. Version: 07 2018

[NAS18c] NASA: *TESS*. https://tess.gsfc.nasa.gov. Version: 07 2018

[NAS18d] NASA: *WFIRST (News Dec. 22, 2017)*. https://www.nasa.gov/wfirst. Version: 07 2018

[Ope18] OPENMP ARCHITECTURE REVIEW BOARD: *OpenMP*. https://www.openmp.org. Version: 07 2018

[Röd17] RÖDER, Frank: *Static Code Analysis for HPC Use Cases*, Universität Hamburg, Bachelor's Thesis, 07 2017

[RR12] Rauber, T. ; Rünger, G.: *Parallele Programmierung*. Springer, 2012 https://www.springer.com/de/book/9783642136030. – ISBN 978–3–642–13604–7

[See08] Seelmann, A. M.: *3D Strahlungstransport - Erste Rechnungen*, Universität Hamburg, Diploma Thesis, 02 2008. https://www.hs.uni-hamburg.de/images/groups/phoenix/diplomarbeit_seelmann.pdf

[Squ16] Squar, Jannek: *MPI-3 algorithms for 3D radiative transfer on Intel Xeon Phi coprocessors*, Universität Hamburg, Master's Thesis, 10 2016. http://edoc.sub.uni-hamburg.de/informatik/volltexte/2017/231/

[TOP18] TOP500: *Statistics*. https://www.top500.org/statistics/list/. Version: 07 2018

[WF92] Wolszczan, A. ; Frail, D. A.: A planetary system around the millisecond pulsar PSR1257 + 12. In: *Nature* 355 (1992), Januar, S. 145–147. http://dx.doi.org/10.1038/355145a0. – DOI 10.1038/355145a0

[Wik18a] Wikipedia: *Empirische Varianz*. https://de.wikipedia.org/wiki/Empirische_Varianz. Version: 07 2018

[Wik18b] Wikipedia: *Methods of detecting exoplanets*. https://en.wikipedia.org/wiki/Methods_of_detecting_exoplanets. Version: 07 2018

[Wik18c] Wikipedia: *Strahlungstransport*. https://de.wikipedia.org/wiki/Strahlungstransport. Version: 07 2018

Anhänge

A. Anpassung der PHOENIX Architektur

Zur Kompilation von PHOENIX auf neuen Systemumgebungen oder mit anderen Compilern, ist es notwendig folgende zwei Dateien anzupassen. In `Source\build` ist lediglich eine Zeile mit dem neuen Architekturstring hinzuzufügen.

Listing A.1: Anpassung des Buildprozesses [`Source/build`]

```
1378 if ( ($altarch ne "HPUX") &&
1379    ...
1531    ($altarch ne "X86_64-FLANG-MPI-OPENBLAS") &&
1532    ...
1655    ){
1656    die("build : error : No recognized architecture flag : $altarch\n");
1657    }
```

In der `Source\Compiler.options` wird der neue Architekturstring dann als Prefix eines neuen Blocks verwendet. Für den hier gezeigten Abschnitt wurde die zuvor vorhandene Architektur `X86_64-GFORTRAN-MPI` übernommen und an die Bibliothek OPENBLAS und den flang Compiler angepasst.

Listing A.2: Anpassung der Architekturen [`Source/Compiler.options`]

```
###############################################################################
# flang for Intel X86_64 and generic MPI using OpenBLAS for blas/lapack
###############################################################################
#X86_64-FLANG-MPI-OPENBLAS # flag for architecture:
#X86_64-FLANG-MPI-OPENBLAS ARCH=gfortran-mpi
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS # code conversion call
#X86_64-FLANG-MPI-OPENBLAS CONVERT=2g95-mpi.pl
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS # General Flags:
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS LAPACK=-lopenblas
#X86_64-FLANG-MPI-OPENBLAS
#X86_64-FLANG-MPI-OPENBLAS LFLAGS= $(OPENCL) $(QDLIB) $(LAPACK)
    $(TOPBINDIR)/cio.o -lstdc++
#X86_64-FLANG-MPI-OPENBLAS
#X86_64-FLANG-MPI-OPENBLAS FLAGS2=
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS # Flags for optimization:
```

```
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS FFLAGS= -O -fno-automatic -fno-second-underscore
#X86_64-FLANG-MPI-OPENBLAS DFLAGS= -c -fno-automatic -fno-second-underscore
#X86_64-FLANG-MPI-OPENBLAS BENCHFLAGS=$(FFLAGS)
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS # Flags for debugging::
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS #FFLAGS= -g -Wall -fno-automatic -fbounds-check
    -fno-second-underscore
#X86_64-FLANG-MPI-OPENBLAS #DFLAGS= -g -Wall -fno-automatic -fbounds-check
    -fno-second-underscore
#X86_64-FLANG-MPI-OPENBLAS
#X86_64-FLANG-MPI-OPENBLAS F95FLAGS=$(FFLAGS)
#X86_64-FLANG-MPI-OPENBLAS F77=mpif90 -fc=flang -m64 CPUOPTS
#X86_64-FLANG-MPI-OPENBLAS F90=mpif90 -fc=flang -m64 CPUOPTS
#X86_64-FLANG-MPI-OPENBLAS LD=mpif90 -fc=flang -m64 CPUOPTS
#X86_64-FLANG-MPI-OPENBLAS CC=mpicc -cc=clang -m64 GCC_CPUOPTS
#X86_64-FLANG-MPI-OPENBLAS
#X86_64-FLANG-MPI-OPENBLAS CFLAGS= -c -DSUN -O -DO_LARGEFILE=0 -DGOODF90
    -DGFORTRAN_CBE
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS # Flags for QD package:
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS CC_OPTS=-DHAS_FMA
#X86_64-FLANG-MPI-OPENBLAS CC_OPTS=
#X86_64-FLANG-MPI-OPENBLAS C_QD_OPTS=-DADD_UNDERSCORE
#X86_64-FLANG-MPI-OPENBLAS CC_QD=g++ -m64 CPUOPTS
#X86_64-FLANG-MPI-OPENBLAS CC_FLAGS=-O4
#X86_64-FLANG-MPI-OPENBLAS AR=ar
#X86_64-FLANG-MPI-OPENBLAS AR_FLAGS=-crv
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS # Flags for SuperLU package:
#X86_64-FLANG-MPI-OPENBLAS #
#X86_64-FLANG-MPI-OPENBLAS ARCH_SLU=ar
#X86_64-FLANG-MPI-OPENBLAS ARCHFLAGS_SLU=-crv
#X86_64-FLANG-MPI-OPENBLAS CC_SLU=gcc -m64 CPUOPTS
#X86_64-FLANG-MPI-OPENBLAS CFLAGS_SLU= -O4 -fstrict-aliasing
    -fno-omit-frame-pointer -funroll-all-loops -frerun-loop-opt
    -DMATH_ASM_INLINE
#X86_64-FLANG-MPI-OPENBLAS FORTRAN=$(F90)
#X86_64-FLANG-MPI-OPENBLAS LOADER=$(LD)
#X86_64-FLANG-MPI-OPENBLAS LOADOPTS=
#X86_64-FLANG-MPI-OPENBLAS CDEFS=-DAdd_
```

PHOENIX lässt sich dann kompilieren mittels `./mk_phoenix X86_64-FLANG-MPI-OPENBLAS` oder `./mk_phoenix -debug X86_64-FLANG-MPI-OPENBLAS` nachdem ggf. die Pfade für flang und OPENBLAS gesetzt wurden.

B. Codesegmente zur Datenaufnahme

Um die Zahl der Charakteristiken zu ermitteln, die pro Raumwinkel (θ, ϕ) anfallen, wurden im Quellcode des Trackers folgende Segmente hinzugefügt. Da die Daten für jede Iteration identisch sind, wurde hier mittels konkreter Werte des Raumwinkels der Beginn einer neuen Iteration erkannt und die Ausgabedatei zurückgesetzt. Dieser Trick war nötig, da weitere Details der umgebenden Schleife über die Raumwinkel innerhalb des Trackers nicht bekannt sind. Sinnvoll ist es bei dieser Datenaufnahme allerdings die Zahl der Iterationen mittels `itlamb = 1` im Jobscript auf eine einzige zu beschränken. Ebenfalls sinnvoll ist es, die Daten mit nur einem einzigen Prozess aufzunehmen, was allerdings im Falle des größeren Gitters nicht möglich war, daher mussten hier die Daten mittels MPI aus mehreren Instanzen zusammengefügt werden. Die mit `cMPI` beginnenden Zeilen werden beim Buildprozess von Kommentarzeilen in Codezeilen gewandelt, sofern die MPI Option aktiv ist. Die Zeilennummern resultieren aus den gezeigten Änderungen basierend auf dem Git Commit 5bcab32 vom 21.03.2018 und sollen nur der groben Einordnung in den bestehenden Code dienen.

Listing B.1: Datenaufnahme zur Gesamtzahl und Gitter-Statistik der Charakteristiken [Source/3DRT/LC_Lstar_tracker_spherical_MultiPassCache.for]

```
41  cMPI     integer :: mpi_rank,mpi_size                              !MPI
42       integer :: debugi,debugj
43       character(len=110) :: debugstr
44       character(len=110), allocatable :: debugarray(:)
45       ...
132 cMPI     call mpi_comm_rank(MyMPI_3DRT_COMM_FS,mpi_rank,ierr)      !MPI
133 cMPI     call mpi_comm_size(MyMPI_3DRT_COMM_FS,mpi_size,ierr)      !MPI
134      ...
695 c--
696 c-- debug output by Oliver Pola:
697 c--
698 c-- remember not only iterating over theta, phi once, also having
        OS_iterations loop
699 c-- dirty trick is to rewind every OS_iteration, to keep only last one
700 c--
701 c-- plot total sum of characteristics for each theta, phi combination
702 c--
703      debugi = 0
704      debugj = 0
705      if((abs(theta - 3.141) .lt. 0.001) .and.
706     &   (abs(phi - 0.0) .lt. 0.001)) then
707       debugi = 1
708      endif
```

```
cMPI     call mpi_reduce(debugi,debugj,1,                          !MPI
cMPI    &              MPI_INTEGER,MPI_MAX,                       !MPI
cMPI    &              0,MyMPI_3DRT_COMM_FS,ierr)                 !MPI
cMPI     if(mpi_rank .eq. 0) then                                  !MPI
cMPI      debugi = debugj                                          !MPI
      if(debugi .eq. 1) then
       rewind(7)
      endif
cMPI      allocate(debugarray(mpi_size))                           !MPI
cMPI     endif                                                     !MPI
      write(debugstr,*) theta, ';', phi, ';', n_chars, ';',
     & sum(chars_pv(:,:,:)), ';', maxval(chars_pv(:,:,:))
cMPI     call mpi_gather(debugstr,110,MPI_CHARACTER,               !MPI
cMPI    &              debugarray,110,MPI_CHARACTER,              !MPI
cMPI    &              0,MyMPI_3DRT_COMM_FS,ierr)                 !MPI
cMPI     if(mpi_rank .eq. 0) then                                  !MPI
      write(7,*) debugstr
cMPI      do debugi=2,mpi_size                                     !MPI
cMPI       write(7,*) debugarray(debugi)                           !MPI
cMPI      enddo                                                    !MPI
cMPI      deallocate(debugarray)                                   !MPI
cMPI     endif                                                     !MPI
```

Alternativ wurden an gleicher Stelle für einen ausgewählten Raumwinkel, hier $(\theta = 1.504, \phi = 1.178)$, in jedem Gitterelement die Zahl der durchlaufenden Charakteristiken ermittelt. Dies hat nur für das kleine Gitter und daher nur mit einem einzelnen Prozess stattgefunden.

Listing B.2: Datenaufnahme zur Zahl der Charakteristiken pro Gitterelement [Source/3DRT/LC_Lstar_tracker_spherical_MultiPassCache.for]

```
c--
c-- debug output by Oliver Pola:
c--
c-- detailed plot of picked theta phi, having number of chars per grid element
c--
      if((abs(theta - 1.504) .lt. 0.001) .and.
     &  (abs(phi - 1.178) .lt. 0.001)) then
       rewind(7)
       do it=-nt,nt
        do ip=-np,np
         do ir=-nr,nr
          write(7,*) ir, ';', it, ';', ip, ';', chars_pv(ir,it,ip)
         enddo
        enddo
       enddo
      endif
```

C. CD-Inhalt

Auf der beigefügten CD befinden sich folgende Daten:

Bachelorthesis_Pola.pdf
: Diese Bachelorthesis im PDF-Format

Charakteristikmessungen/
: Messdaten zur Anzahl Charakteristiken in Abhängigkeit vom Raumwinkel

Jobs/
: Zur Messung verwendete Jobscripts

Jobs/Ergebnisse.tar.gz
: Eine kleine Stichprobe der von den Jobs erzeugten Ausgaben

Laufzeitmessungen/
: Messdaten zur Laufzeit der Jobs

Laufzeitmessungen/allruns.csv
: Tabelle aller Einzelmessungen

Laufzeitmessungen/iteration*.csv
: Selektion und Mittelwertbildung aus den Einzelmessungen

Plots
: Aus den Messdaten generierte Abbildungen
(die *.plt sollten, mit gnuplot geöffnet, interaktiv bedienbar sein)

Profiles
: Durch Score-P und perf erzeugte Profile und Spurdaten

Programmanpassungen
: Die im Rahmen dieser Arbeit in PHOENIX angepassten Dateien

Die verwendeten Dateinamen setzen sich aus folgenden Teilen zusammen:

small Verwendung des kleinen 17×17×65 Gitters, ohne Tracking Cache

cacheOFF Verwendung des großen 257×33×65 Gitters, ohne Tracking Cache

cacheON Verwendung des großen 257×33×65 Gitters, mit Tracking Cache

west, hummel, etc. Bezeichnung des Systems

INTEL, GFORTRAN, FLANG Verwendung des angegeben Compilers

nX, pX, tX Zusammengefasste Messdaten, bei denen jeweils die Zahl der Nodes, Prozesse oder Threads auf X festgehalten ist

original PHOENIX im Zustand des Git Commits 5bcab32 vom 21.03.2018

ipitir Angepasstes PHOENIX mit Schleifenreihenfolge ip, it, ir

-itipir Schleifenreihenfolge it, ip, ir, wobei it mit Schrittweite -1 durchlaufen wird

~itipir Schleifenreihenfolge it, ip, ir, wobei die Schrittweite -1 oder 1 in it abhängig von θ gewählt ist

itipir+1 Schleifenreihenfolge it, ip, ir, wobei ir mit Offset +1 durchlaufen wird

openmp PHOENIX mit angepasstem OpenMP-Schema

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Computing in Science 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.

Hamburg, den 06.08.2018 Oliver Pola

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 06.08.2018 Oliver Pola