Untersuchungen zu schnelleren Berechnungsverfahren innerhalb von ARTM mithilfe von Parallelisierung

GRS - 820

Gesellschaft für Anlagen-
und Reaktorsicherheit
(GRS) gGmbH

Untersuchungen zu schnelleren Berechnungsverfahren innerhalb von ARTM mithilfe von Parallelisierung

Abschlussbericht

Karsten Spieker
Robert Hanfland

Oktober 2025

Anmerkung:

Das diesem Bericht zugrunde liegende Eigenforschungsvorhaben wurde mit Mitteln des Bundesministeriums für Umwelt, Klimaschutz, Naturschutz und nukleare Sicherheit (BMUKN) unter dem Förderkennzeichen 3624S72523 durchgeführt.

Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei der GRS.

Der Bericht gibt die Auffassung und Meinung der GRS wieder und muss nicht mit der Meinung des BMUKN übereinstimmen.

GRS - 820
ISBN 978-3-911727-13-6

Deskriptoren

ARTM, Atmosphärische Ausbreitung, CPU, GPU, Parallelisierung, verteiltes Rechnen

Kurzfassung

Dieses Eigenforschungsvorhaben hatte den Kompetenzaufbau der GRS bezüglich der nachträglichen Parallelisierung komplexer Simulations- und Modellierungsprogramme zum Ziel. Der Kompetenzaufbau erfolgte beispielhaft am Simulationsprogramm ARTM, da dafür bereits eine breite Expertise des Programmablaufs bei der GRS besteht. Nach einer Diskussion über die grundlegenden parallelen Hardware-Architekturen, den in anderen Atmosphärischen Ausbreitungsmodellen verwendeten Parallelisierungsstrategien und einigen gängigen parallelen Programmiermodellen wurde der Quellcode analysiert, um geeignete Stellen für eine Parallelisierung in ARTM und TALdia zu identifizieren. Aus den gewonnenen Erkenntnissen wurde ein Konzept zur Parallelisierung der Simulationsprogrammbestandteile erarbeitet und umgesetzt. Aufgetretene Probleme und Besonderheiten, die bei der Umarbeitung eines sequenziellen in ein paralleles Programm auftraten, wurden diskutiert. Aus den Ergebnissen der Umarbeitung ließ sich der prognostizierte Anpassungsbedarf für komplexe Simulations- und Modellierungsprogramme im Allgemeinen und der für ARTM und TALdia im Speziellen ableiten. Hauptaugenmerk ist demnach auf die Verwendung von für den parallelen Programmablauf geeignete Speicherstrukturen und die Einhaltung des EVA-Prinzips zu legen. Die erfolgte Parallelisierung von ARTM und TALdia ergab eine reale Verkürzung der Rechenzeit um einen Faktor 3 – 4 auf einem handelsüblichen Laptop.

Abstract

This research project aimed to systematically enhance the expertise of the GRS in the parallelization of complex simulation and modeling software. The ARTM simulation program was selected as a representative case study, given GRS's extensive prior knowledge of its internal structure and computational workflow. The project commenced with a comprehensive review of fundamental parallel hardware architectures, parallelization strategies employed in other atmospheric dispersion models, and widely used parallel programming paradigms. Subsequently, the source code of ARTM and TALdia was analyzed to identify code segments amenable to parallel execution. Based on the insights gained, a structured concept for the parallelization of key components within the simulation programs was developed and implemented. Challenges and specific issues encountered during the transformation from a sequential to a parallel program were systematically examined and documented.

The outcomes of this transformation enabled the derivation of general adaptation requirements for complex simulation and modeling software, as well as specific recommendations for ARTM and TALdia. Particular emphasis was placed on the use of memory structures suitable for parallel execution and strict adherence to the input–processing–output (IPO) principle. The implemented parallelization of ARTM and TALdia resulted in a reduction in computation time by a factor of 3 to 4 on a standard consumer-grade laptop.

Inhaltsverzeichnis

1 Einleitung

Die steigende Komplexität von Simulation und Modellierung im Bereich der Forschung, z. B. in der numerische Strömungsmechanik oder auch der Dispersionsmodellierung, sowie das weitere Voranschreiten von maschinellem Lernen und künstlichen Intelligenzen geht stets mit einer erhöhten Programmlaufzeit einher. Aus diesem Grund ist es unabdingbar, schnellere Berechnungsverfahren zu verfolgen. In den letzten Jahren wurde auch das „General Purpose Computation on Graphics Processing Unit“ (GPGPU) intensiv verfolgt. Weitere Berechnungsverfahren zum Beschleunigen der Laufzeit sind die Parallelisierungen über Mehrkernprozessoren, den heute üblichen zentralen Berechnungseinheiten (engl. „central processing units“ (CPUs)), sowie das verteilte Rechnen.

Mit dem Ziel, die Kompetenz der GRS im Bereich der Parallelisierung von Simulations- und Modellcodes zu erweitern, sollen deshalb exemplarisch anhand des ARTM (Atmosphärisches Radionuklid Transport-Modell) die gängigsten Parallelisierungsmöglichkeiten untersucht werden. Die Untersuchung mittels ARTM eignet sich aufgrund dessen Entwicklung als Open Source-Projekt. Ebenso wird ARTM vom Bundesamt für Strahlenschutz (BfS) verwendet und einem breiten Kreis von Anwenderinnen und Anwendern zur Verfügung gestellt. Der Kompetenzaufbau der GRS in der Weiterentwicklung und Anwendung von ARTM ist somit von großem Interesse für die Allgemeinheit. Des Weiteren basiert ARTM auf einem Lagrange-Partikelmodell und eignet sich insbesondere durch den Transport von Simulationspartikeln sehr gut für die Untersuchung der unterschiedlichen Berechnungsverfahren. Zuletzt wird für den sehr komplexen Einbau der Parallelisierungsmöglichkeiten verlangt, dass ein sehr gutes Verständnis in die innere Struktur des zu parallelisierenden Programms vorliegen muss.

Im Eigenforschungsvorhaben wird deshalb der internationale Stand von Wissenschaft und Technik zur Parallelisierung von Lagrange-Partikelmodellen oder ähnlichen Ausbreitungsmodellen recherchiert und bewertet. Des Weiteren wird eine Recherche zu hardwarenahen Parallelisierungsmöglichkeiten und dazu passenden Bibliotheken, Kompilierern und Grafikkarten (bei GPU-Parallelisierung) durchgeführt. Basierend auf diesen Erkenntnissen sowie der Sicherstellung der Übergabe aller notwendigen Informationen an weitere Prozesse, werden mögliche Stellen innerhalb des Quellcodes von ARTM für eine Parallelisierung identifiziert. In einem letzten Schritt werden mögliche Implementierungsmethoden und dafür notwendige Anpassungen in ARTM analysiert. Ziel ist, das Grundkonzept von ARTM so wenig wie möglich anzupassen, so dass die über die Jahre aufgebaute Kompetenz erhalten bleibt.

Dieser Abschlussbericht ist in folgende Teile unterteilt. In Kapitel 2 werden einige wichtige Fachbegriffe erläutert, die in diesem Bericht Verwendung finden. Anschließend werden in Kapitel 3 grundlegende Rechner- und Speicherarchitekturen diskutiert, die für die Konzepte der parallelen Programmierung von Bedeutung sind. In Kapitel 4 werden verschiedene parallele Programmiermodelle vorgestellt, die für die Parallelisierung von ARTM in Betracht kommen könnten. Kapitel 5 beschäftigt sich mit der Identifikation möglicher Stellen innerhalb des Quellcodes von ARTM/TALdia für eine Parallelisierung, während das Kapitel 6 in Detail mögliche Implementierungsmethoden und erste Ergebnisse einer parallelisierten Version von ARTM/TALdia präsentiert. Anschließend werden die Ergebnisse des Eigenforschungsvorhabens in Kapitel 7 zusammengefasst.

2 Begriffserklärung

Tab. 2.1 Überblick über relevante Begriffe innerhalb dieses Dokuments

Begriff deutsch	Begriff englisch	Erklärung/Bedeutung	Quelle
Teilaufgaben		Eine auszuführende Aufgabe (z. B. Matrixmultiplikation) wird in Teilaufgaben zerlegt.	/RAU 07/
Prozesse, Threads, Tasks	Processes, Threads, Tasks	Teilaufgaben werden als Prozesse, Threads oder Tasks bezeichnet, wobei sich die Bedeutungen für unterschiedliche Programmiermodelle und -umgebungen geringfügig unterscheiden können. Thread Meist als Bezeichnung für Teilaufgaben verwendet, die auf einen *gemeinsamen Adressraum* zugreifen. Prozess Meist als Bezeichnung für Teilaufgaben verwendet, die auf den *privaten* Bereich eines *verteilten Adressraums* zugreifen.	/RAU 07/
Prozessor	Processor	Eine unabhängige physikalische Berechnungseinheit. Das können sowohl Prozessoren sein als auch Prozessorkerne bei Multi-Prozessoren.	/RAU 07/
Berechnungsstrom		Das ist eine Teilaufgabe.	/RAU 07/
Mapping	Mapping	Mapping ist das Abbilden von Teilaufgaben auf physikalische Berechnungseinheiten.	/RAU 07/
Synchronisation	Synchronisation	Prozess zum Informationsaustauch zwischen Berechnungsströmen (Teilaufgaben).	/RAU 07/
Kostenmaß		Maß zur Bewertung der Ausführungszeit eines parallelen Programms	/RAU 07/

Begriff deutsch	Begriff englisch	Erklärung/Bedeutung	Quelle
Parallele Laufzeit	Parallel run time	Die parallele Laufzeit setzt sich aus der Zeit, während der Berechnungsströme parallel ablaufen, und der Zeit für Informationsaustauch/Synchronisation zusammen.	/RAU 07/
Lastgleichgewicht	Load balancing	Verteilung der Rechenlast auf die beteiligten Prozessoren	/RAU 07/
Granularität	Granularity	Durchschnittliche Größe der Teilaufgaben, in die eine Aufgabe zerlegt wird (z. B. gemessen als Anzahl der Instruktionen)	/RAU 07/
Scheduling	Scheduling	Entscheidungsvorgang, in welcher Reihenfolge die Teilaufgaben, unter Berücksichtigung von Abhängigkeiten, abgearbeitet werden	/RAU 07/
UMA-System (Uniform Memory Access)	UMA-System (Uniform Memory Access)	Einheitliche Speicherzugriffszeit für alle Prozessoren bei einer gemeinsamen Speicher-Architektur („shared memory“). Häufigster Vertreter ist der Symmetric Multiprocessor (SMP) Rechner.	/RAU 07/ /BAR 17/
NUMA-System (Non-Uniform Memory Access)	NUMA-System (Non-Uniform Memory Access)	Die Speicherzugriffszeit eines Prozessors hängt von der relativen Speicherstelle ab.	/RAU 07/ /BAR 17/
Verschränktes Multithreading	Interleaved Multithreading	Ein physikalischer Prozessor wird in eine feste Anzahl virtueller Prozessoren unterteilt. Nach jedem Maschinenbefehl findet ein expliziter Wechsel zum nächsten virtuellen Prozessor statt. Dadurch führt gleichzeitig zur Speicherzeit des ersten virtuellen Prozessors der zweite virtuelle Prozessor seinen Maschinenbefehl aus, usw.	/RAU 07/

Begriff deutsch	Begriff englisch	Erklärung/Bedeutung	Quelle
Simultanes Multithreading (SMT)	Simultaneous Multithreading (SMT)	Ein physikalischer Prozessor ist hardwareseitig in logische Prozessoren unterteilt, d. h. bestimmte Steuerungseinheiten des Prozessors sind mehrfach vorhanden. Threads laufen auf einem logischen Prozessor. Fällt ein Thread/logischer Prozessor in einen Wartezustand, etwa wegen Kommunikation mit dem Speicher, übernimmt ein anderer Thread/logischer Prozessor (wird auch als Hyperthreading bezeichnet).	/RAU 07/
Cache-Speicher	Cache	Kleine, schnelle Speicher, die zwischen Prozessor und Hauptspeicher geschaltet sind	/RAU 07/
Cache-Kohärenz	Cache coherence	Speicher- bzw. Cache-Kohärenz liegt vor, wenn jeder Prozessor das *gleiche eindeutige* Bild des Speichers hat.	/RAU 07/
Rechnersystem	Computer system	Gesamtheit der Hard- und Software, die dem Programmierer zur Verfügung steht und seine „Sicht“ auf den Rechner bestimmt	/RAU 07/
Zentraleinheit	Central Processing Unit (CPU)	Die eigentliche Berechnungseinheit. Die CPU enthält zwei weitere Hauptkomponenten, nämlich die Kontrolleinheit (engl. „control unit“) und die „arithmetic logic unit“ (ALU).	/RAU 07/
Knoten	Node	Ein eigenständiger „Computer in der Box“. Knoten sind miteinander vernetzt und bilden einen Supercomputer.	/BAR 17/
Kontrolleinheit	Control unit	Die Kontrolleinheit sorgt dafür, dass die Maschinenbefehle und Daten aus dem Speicher geladen werden und koordiniert deren sequenzielle Abarbeitung in der ALU.	/RAU 07/

Begriff deutsch	Begriff englisch	Erklärung/Bedeutung	Quelle
Arithmetisch-logische Einheit	Arithmetic logic unit (ALU)	Die ALU führt entsprechend den Maschinenbefehlen grundlegende arithmetische Operationen mit den geladenen Daten durch.	/RAU 07/
Stream-Verarbeitung	Stream processing	Bei der Stream-Verarbeitung handelt es sich um die kontinuierliche Verarbeitung neuer Datenereignisse, sobald diese empfangen werden.	/NVI 24a/
Prozessorkern	Processing core	Eine unabhängige Berechnungseinheit, die die wesentlichen Elemente zur Abarbeitung von Maschinenbefehlen (Kontrolleinheit und Arithmetisch-logische Einheit) enthält. Bei einem Ein-Kern-Prozessor stellt im Wesentlichen die CPU den Kern dar. Bei Mehrkernprozessoren sind auf der CPU mehrere Kerne mit einer zugehörigen Steuereinheit enthalten.	
Adressraum	Address space	Die Menge aller zulässigen Adressen im gesamten Speicher, die einer Anwendung zugeordnet sind	
Programmiermodell	Programming model	Die Gesamtheit aller Grundprinzipien und Schnittstellen, über die der Entwickler die Ressourcen der Maschine ansteuern kann. Sie definiert also die Sicht des Programmierers auf die Maschine und wie der Programmierer diese ansprechen kann.	
Overhead	Overhead	Die Gesamtheit aller zusätzlichen Arbeiten, die dadurch entstehen, dass ein Programm nicht mehr sequenziell, sondern parallel abläuft	/UNI 20/
Amdahlsches Gesetz	Amdahls law	Das Amdahlsche Gesetz beschreibt den Zusammenhang zwischen den sequenziellen und parallelen Anteilen sowie der Anzahl der *Prozessorkerne* und der potenziellen Beschleunigung eines Programms.	/TRO 18/,/RAU 07/

Begriff deutsch	Begriff englisch	Erklärung/Bedeutung	Quelle
Aufrufstapel	Call stack	Der Aufrufstapel, oft auch nur als „Stack“ bezeichnet, ist ein Speicherbereich für Funktionen im Hauptspeicher. Hier werden nacheinander die Funktionen, ihre Variablen und Anweisungen abgelegt. Die Funktionen, die als letztes begonnen wurden, werden als erstes wieder beendet.	/WOL 19/
Allokation	allocation	Dt.: Zuweisung. Bei einer Speicherallokation wird einer Variablen ein Speicherbereich dynamisch, d. h. zur Laufzeit, zugewiesen.	/WOL 19/
Struktur	struct	In einer Struktur können Variablen mit verschiedenen Datentypen zusammengefasst werden. Dies unterscheidet sie von Feldern (engl. „arrays“), bei denen nur Variablen desselben Datentyps verwendet werden.	/WOL 19/
Verkettete Liste	Linked list	Eine verkettete Liste ist eine Datenstruktur, in der verschiedene Datentypen dynamisch, d. h. zur Laufzeit, gespeichert werden können. Dabei ist jeder Listeneintrag durch seine Nachbarn, also den Vorgänger und den Nachfolger, eindeutig in der Liste eingeordnet.	/HAK 04/
Stalls	Stalls	Moderne CPUs führen Befehle nicht strikt nacheinander, sondern in einer Pipeline und oft out-of-order aus. Im Idealfall ist die Pipeline voll, und jede Taktung erledigt nützliche Arbeit. Ein Stall tritt auf, wenn die Pipeline wartet, weil eine Ressource fehlt oder irgendeine Abhängigkeit nicht erfüllt ist. In solchen Fällen taktet der Kern, aber macht keine Arbeit.	/INT 25a/
Retiring	Retiring	Das „Retiring“ ist der prozentuale Anteil der Takte, in denen die CPU nützliche Instruktionen fertig ausgeführt hat.	/INT 25a/

3 Arbeitspaket 1: Recherche zu hardwarenahen Parallelisierungsmöglichkeiten

Parallelisierung kann auf verschiedenste Weisen klassifiziert werden, die auf unterschiedlichen Eigenschaften der Rechnersysteme, also der Kombination aus Hardware- und Softwarekomponenten, beruhen. Das können z. B. die Flynnsche Klassifizierung[1], die auf einer Quantifizierung von Befehls- und Datenströmen basiert, die Speicherorganisation oder die Verbindungsnetzwerkklassifikation sein /RAU 07/. Im Folgenden wird keine bestimmte Klassifizierung zur Strukturierung verwendet. Stattdessen werden wichtigen Architektur- und Bauteileigenschaften berücksichtigt, die in der praktischen Anwendung der parallelen Programmierung von Bedeutung sind. Daher wird in den folgenden Abschnitten auf Systeme mit gemeinsamem Speicher (engl. „shared memory systems“), Systeme mit verteiltem Speicher (engl. „distributed systems“) und Grafikprozessoreinheiten (engl. „graphic processor units“ (GPU)) eingegangen /TRO 18/.

Darüber hinaus werden typische Aufgaben genannt, die für eine Parallelisierung auf den unterschiedlichen Systemen besonders geeignet sind. Es gibt allerdings auch Aufgaben, die gar nicht parallelisierbar sind. Dabei handelt es sich um Aufgaben, die von den Ergebnissen der vorherigen Berechnung abhängen, wie es z. B. bei manchen iterativen Algorithmen der Fall ist. Diese Probleme weisen eine Zeitabhängigkeit oder Kausalität auf und sind daher nicht parallelisierbar /NAV 14/.

3.1 Grundlegende Rechnerarchitektur

Heutzutage funktionieren die meisten Prozessoren nach der Von-Neumann Architektur /COO 12/, /BAR 17/. Dieser Aufbau besteht im Wesentlichen aus drei Teilen, dem Speicher, der eigentlichen Zentraleinheit oder zentralen Berechnungseinheit (engl. „central processing unit“ (CPU)) und einer Schnittstelle für die Ein- und Ausgabe. Eine schematische Darstellung ist in Abb. 3.1 abgebildet. Die CPU enthält zwei weitere Hauptkomponenten, nämlich die Kontrolleinheit (engl. „control unit“) und die arithmetisch-logische Einheit (engl. „arithmetic logic unit“ (ALU)). Maschinenbefehle und Daten sind in einem elektronischen Speicher abgelegt. Die Kontrolleinheit sorgt dafür, dass die Maschinen-

[1] Theoretische Klassifizierung, die am Anfang der Parallelrechnungsentwicklung stand. Rechner werden in Single Instruction Single Data (SISD), Multiple Instructions Single Data (MISD), Single Instruction Multiple Data (SIMD) und Multiple Instructions Multiple Data (MIMD) unterteilt /RAU 07/.

befehle und Daten aus dem Speicher geladen werden und koordiniert deren sequenzielle Abarbeitung in der ALU. Die ALU führt entsprechend den Maschinenbefehlen grundlegende arithmetische Operationen mit den geladenen Daten durch. Die Ein- und Ausgabe stellt die Schnittstelle zum Menschen dar. Auch parallele Rechner folgen immer noch diesem grundlegenden Aufbau, allerdings mit einer Vielzahl von Berechnungseinheiten /BAR 17/.

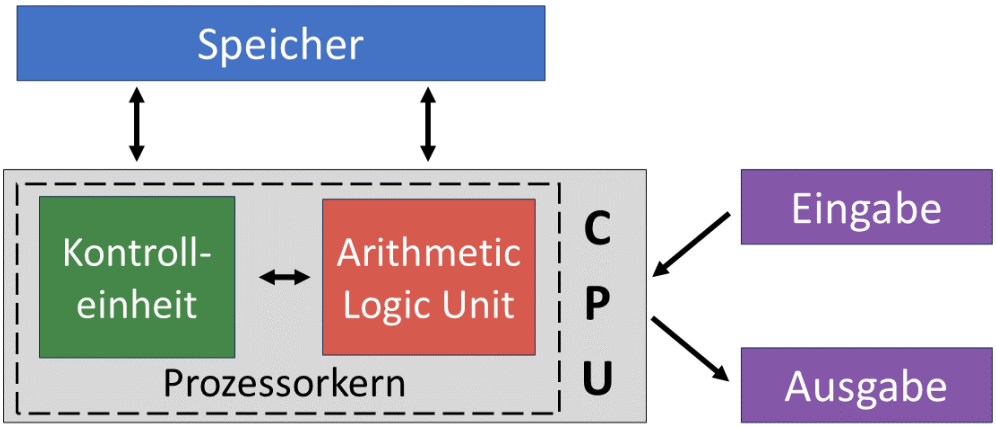


Abb. 3.1 Schematische Darstellung einer zentralen Berechnungseinheit nach der Von-Neumann Architektur

In der Abbildung ist die zentrale Berechnungseinheit (CPU) mit der enthaltenen Kontrolleinheit und der arithmetisch-logische Einheit (engl. „arithmetic logic unit“ (ALU)) dargestellt. Der elektronische Speicher tauscht Informationen mit der Kontrolleinheit und der ALU aus. Die Ein- und Ausgabe stellt die Schnittstelle zum Menschen dar. In der Darstellung stellt die Einheit aus Kontrolleinheit und die ALU den Prozessorkern dar /BAR 17/.

3.2 CPU-Parallelisierung

Unter CPU-Parallelisierung versteht man die Parallelisierung durch Multicore-Prozessoren. Dabei werden mehrere Prozessorkerne auf einem Prozessorchip platziert. Jeder Prozessorkern stellt einen logischen Prozessor dar, der separat z. B. vom Betriebssystem oder einem Programm angesteuert werden kann /RAU 07/.

Solche Multicore-Prozessoren werden als „shared memory“ Systeme bezeichnet /TRO 18/, sie besitzen also einen gemeinsamen Speicher, den sich alle Prozessorkerne teilen. In der Praxis wird allerdings häufig jedem Prozessorkern ein bestimmter Cache-Speicher (L1-Cache) privat, d. h. exklusiv, zugeordnet. Durch den gemeinsamen Speicher sind Änderungen an einer Speicheradresse automatisch für alle Prozessorkerne sichtbar. Dabei kann die genaue Speicherarchitektur bei unterschiedlichen Multicore-

Prozessoren variieren (hierarchisches, Pipeline- oder netzwerkbasiertes Design). Die häufigsten Multicore-Prozessoren gehören zur Familie der symmetrischen Multiprozessoren (SMPs) und sind nach dem „Uniform Memory Access“ (UMA)-System entworfen. Eine schematische Darstellung dieser Speicherarchitektur ist in Abb. 3.2 gegeben /RAU 07/, /BAR 17/. Hierbei haben die Prozessorkerne einen physikalisch gemeinsamen Speicher. Das UMA-System wird manchmal auch als Cache Kohärentes UMA-System (engl. „Cache Coherent-UMA“ (CC-UMA)) bezeichnet. Die Cache-Kohärenz wird hardwareseitig realisiert und bedeutet, dass jeder Prozessorkern zu jeder Zeit dasselbe Speicherabbild sieht, wie alle anderen Prozessorkerne. Das bezieht auch die privaten Cache-Speicher mit ein. Durch diese Speicherarchitektur ist sichergestellt, dass jeder Prozessorkern die gleichen Speicherzugriffszeiten aufweist /BAR 17/.

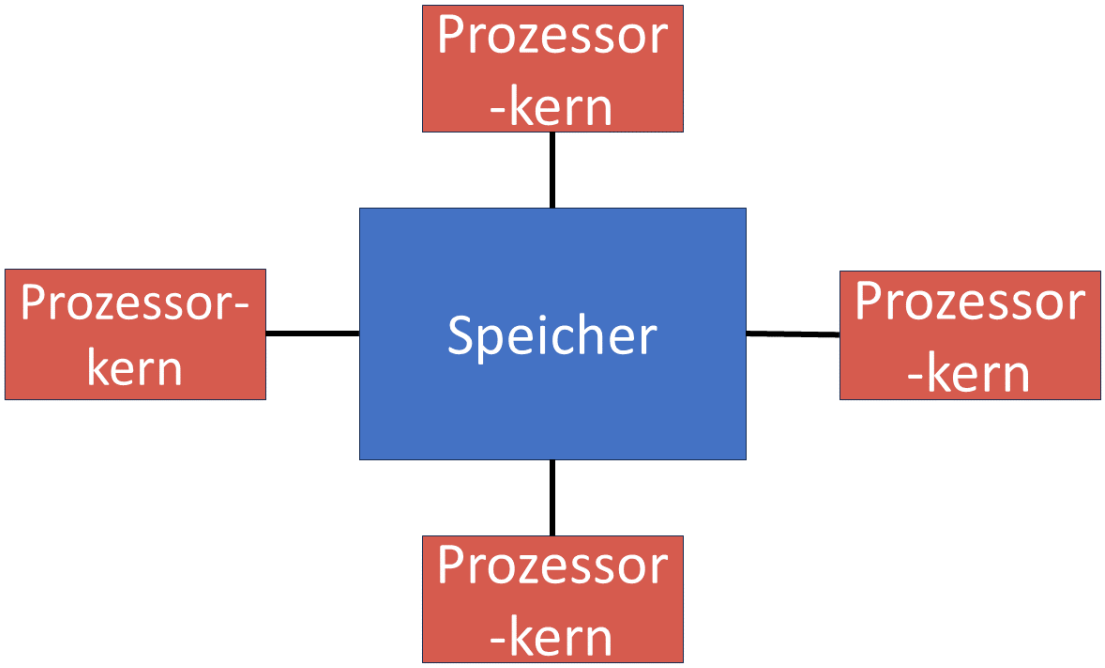


Abb. 3.2 Schema eines UMA-Systems als Repräsentant der „shared memory” Architektur

Alle Prozessorkerne haben einen gemeinsamen Speicher /BAR 17/.

Eine weitere Speicherarchitektur von Multicore-Prozessoren stellt das „Non-Uniform Memory Access“ (NUMA)-System dar. Hierbei wird der physikalisch gemeinsame Speicher durch Verbindungsleitungen zwischen Speicherbereichen realisiert /BAR 17/. Dabei befinden sich alle Prozessorkerne und Speicherbereiche immer noch auf demselben Chip /RAU 07/. Diese Speicherarchitektur ist in Abb. 3.3 dargestellt. Multicore-Prozessoren dieser Architektur werden oft durch die Kombination von zwei oder mehr SMPs realisiert. Ein Merkmal dieser Speicherarchitektur ist, dass nicht alle Prozessorkerne die gleichen

Speicherzugriffszeiten haben, je nachdem, wo die gewünschte Information gespeichert ist. Der Speicherzugriff über die Verbindungsleitung ist langsamer als der Speicherzugriff zum lokalen Speicher. Cache-Kohärenz ist nicht automatisch gegeben. Systeme, die diese bieten, werden als „Cache Coherent-NUMA“ (CC-NUMA) -Systeme bezeichnet /BAR 17/.

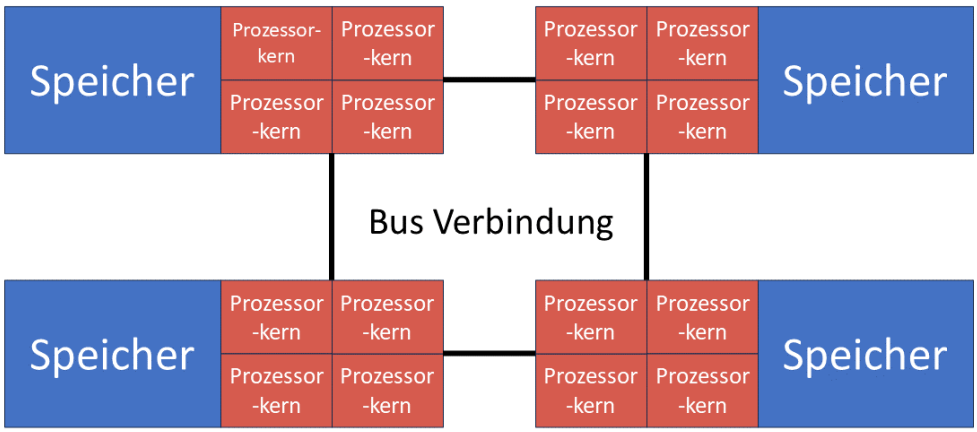


Abb. 3.3 Schema eines NUMA-Systems als Repräsentant der “shared memory” Architektur

Mehrere Prozessor-Speichereinheiten werden durch eine Verbindungsleitung auf einem Chip verbunden /BAR 17/.

Die parallele Programmierung mit Prozessoren bietet eine Reihe von Vorteilen. Durch den globalen, gemeinsamen Adressraum ergibt sich eine benutzerfreundliche Perspektive auf den Speicher. Zudem läuft das Teilen von Daten schnell und einheitlich ab. Speicherkonflikte, wie der schreibende Zugriff von zwei Prozessorkernen auf dieselbe Speicheradresse, werden auf der Ebene der Programmiermodelle (siehe Kapitel 4) verhindert. Außerdem wird der Speicherplatz effizient ausgenutzt, da das Duplizieren von Daten wegen des gemeinsamen Speichers kaum nötig ist /RAU 07/, /BAR 17/. Ein Nachteil dieser Speicherarchitektur ist, dass sie nicht beliebig skalierbar ist. Der Datenverkehr zwischen dem Prozessorkern und dem Speicher kann überproportional zur Anzahl der Prozessorkerne steigen. Zudem stellt die Cache-Kohärenz bei Systemen mit vielen Prozessorkernen eine Herausforderung dar /RAU 07/, /BAR 17/.

Zur Umsetzung der Parallelisierung mit Multicore-Prozessoren werden threadbasierte Programmiermodelle angewendet /TRO 18/. Außerdem sind für Multicore-Prozessoren besonders task-parallele Aufgaben geeignet. Bei Ihnen besteht die zu erfüllende Aufgabe aus Teilaufgaben, die unterschiedliche Funktionen darstellen. Um die Gesamtaufgabe zu erfüllen, müssen alle Teilaufgaben/Funktionen auf einen gemeinsamen Strom

von Daten angewendet werden, ohne dass es eine Zeitabhängigkeit gibt. Grund für die besondere Eignung solcher Probleme ist die Architektur von CPUs, die auf jedem Prozessor bzw. Prozessorkern die unabhängige Ausführung einer eigenen Teilaufgabe erlaubt /NAV 14/.

3.3 GPU-Beschleunigung

Um die Laufzeit von rechenintensiven Anwendungen zu beschleunigen, können in vielen Fällen spezielle Bauteile verwendet werden. Ein Vertreter solcher Bauteile sind Grafikkarten. Diese haben zu Beginn des 21. Jahrhunderts den Sprung geschafft von auf Grafikanwendungen spezialisierten Bauteilen hin zu allgemein anwendbaren Grafikprozessoreinheiten /COO 12/.

Die Architektur von modernen GPUs ähnelt im Kleinen sehr stark der Architektur von modernen Cluster-Systemen im Großen. Eine schematische Darstellung einer GPU, eingebettet in einem Wirtsystem, ist in Abb. 3.4 gezeigt. Bei GPUs werden die Berechnungseinheiten als „streaming multiprocessors“ (SMs) bezeichnet, von denen typischerweise mehrere auf einer Grafikkarte platziert sind. Jeder SM verfügt über einen eigenen L1-Cache, allerdings teilen sich alle SMs einen gemeinsamen L2-Cache, der auch als Datenverbindung zwischen den verschiedenen SMs fungiert. Größere Datenmengen werden in einem nachgeordneten globalen Speicher gehalten. Dieser Speicher ist über eine „Peripheral Component Interconnect Express“ (PCI-E) -Schnittstelle mit dem PCI-E Switch des Wirtsystems verbunden. Auf diese Weise erfolgt sowohl die Kommunikation zwischen GPU und Wirtsystem, als auch die Kommunikation zwischen mehreren GPUs /COO 12/. In einem SM sind mehrere „stream processors“ (SP) zusammengefasst. Wenn ein SM seine Arbeit aufnimmt, dann wird die gleiche Maschinenbefehlssequenz synchron von allen enthaltenen SPs ausgeführt, wobei jeder SP auf einer/m anderen Information/Datum operiert. Das Verhalten entspricht also einer Vektorverarbeitungseinheit (engl. „vector-processing unit“) und ist analog zum SIMD-Prinzip der Flynnschen Klassifizierung /BAR 23/.

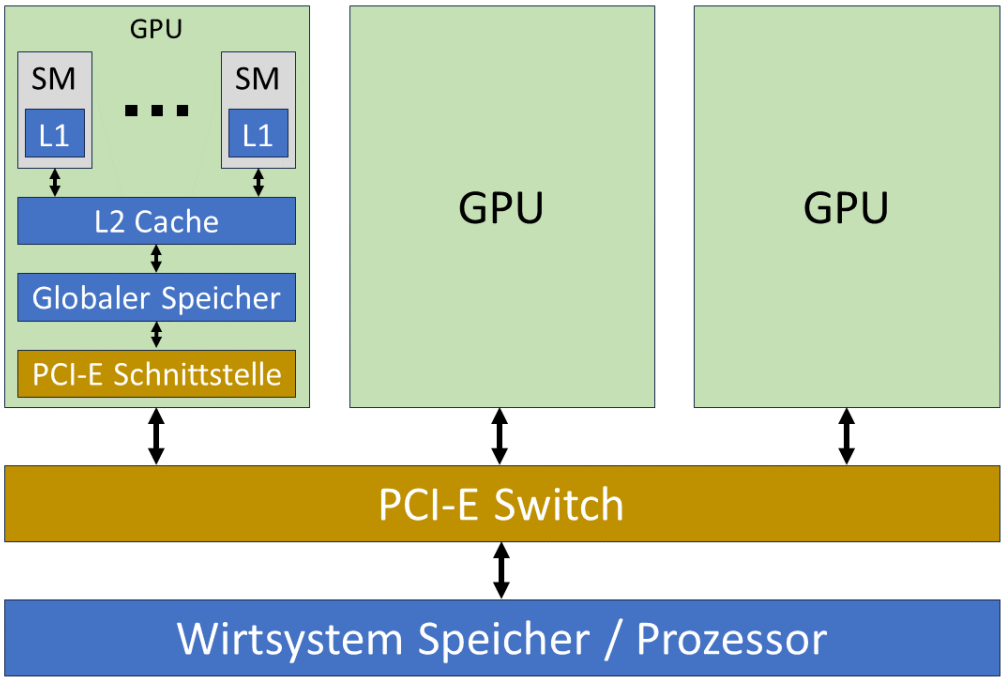


Abb. 3.4 GPU-Architektur eingebettet in ein Wirtsystem mit drei GPUs

Das Wirtsystem ist über ein Verbindungsnetzwerk (PCI-E) mit drei GPUs verbunden. Analog zur Hierarchie der GPUs im Wirtsystem sind auch die einzelnen GPUs aufgebaut. Ein globaler Speicher in der Grafikkarte ist mit einem L2 Cache verbunden. Dieser übernimmt neben der Funktion eines Speichers gleichzeitig auch noch die Funktion eines Verbindungsnetzwerks. Am oberen Ende der Hierarchie stehen mehrere streaming multiprocessors“ (SMs), die jeweils über einen eigenen L1 Cache (L1) verfügen.

Der große Vorteil bei der Verwendung von GPUs ist die hohe Rechenleistung, die durch die massive parallele Architektur entsteht. Allerdings ist die Datenhaltung und Partitionierung[2] dem Programmierer überlassen. Zudem muss die Struktur des zu lösenden Problems bzw. der Algorithmus für die Berechnung durch eine GPU geeignet sein, was im Allgemeinen nicht der Fall ist /COO 12/, /PLL 17/.

Um GPUs zur parallelen Berechnung nutzen zu können, sind sogenannte Streaming basierte Programmiermodelle erforderlich. Besonders geeignet für GPUs sind Berechnungen, die sich natürlicherweise in viele unabhängige Teilberechnungen unterteilen lassen, wie z. B. Operationen auf Matrizen oder Probleme aus der Linearen Algebra. Viele solcher Probleme werden als datenparallel bezeichnet. Eine Teilaufgabe wird dabei auf viele unabhängige Daten angewendet. Man spricht bei solchen Aufgaben auch

[2] Das zerteilen einer Aufgabe in sinnvolle Teilaufgaben.

von „embarrassingly parallel“[3] Aufgaben. Datenparallele Probleme sind ideale Kandidaten für GPUs. Der Grund dafür liegt in der Architektur einer GPU. Sie arbeitet am besten, wenn alle Threads dieselben Befehle abarbeiten, aber auf unterschiedliche Daten /NAV 14/.

3.4 Verteiltes Rechnen

Unter verteiltes Rechnen (engl. „distributed computing“) wird im Allgemeinen verstanden, dass Teilaufgaben einer zu lösenden Aufgabe auf verteilten Systemen (engl. „distributed systems“) bearbeitet werden. Dieser Aufbau stellt den einfachsten Fall dar, um ein System für das parallele Rechnen bereitzustellen /RAU 07/, /TRO 18/.

In der ursprünglichen Konfiguration wird bei verteilten Systemen die „distributed memory“ Architektur verwendet /TRO 18/. Die genauen Ausprägungen dieser Architektur können variieren, es haben jedoch alle gemeinsam, dass für die Interprozessorkommunikation ein Kommunikationsnetzwerk verwendet wird, wie z. B. Ethernet. Die Abb. 3.5 zeigt eine schematische Darstellung der „distributed memory“ Architektur. Jeder Prozessor verfügt über einen lokalen Speicher mit einem eigenen Adressraum. Ein globaler Adressraum, wie bei NUMA-Systemen (vgl. Abb. 3.3), existiert nicht. Da jede Einheit aus Prozessor und Speicher unabhängig von den anderen ist, gibt es keine Cache-Kohärenz. Die Kommunikation zwischen den verschiedenen Einheiten aus Prozessor und Speicher muss vom Programmierer übernommen werden /BAR 17/.

Praktisch lässt sich ein solches System bereits dadurch realisieren, dass mehrere herkömmliche Desktop PCs mit Ein-Kern-Prozessoren über ein Ethernet-Verbindungsnetz miteinander verbunden werden. Dies wurde auch in den Anfängen des parallelen Rechnens so praktiziert /COO 12/.

[3] Der Ausdruck „embarrassingly parallel“ (dt. peinlich parallel) bedeutet, dass es peinlich wäre, wenn man die Vorteile der Parallelisierung nicht für eine Aufgabe oder Teilaufgabe nutzen würde /NAV 14/.

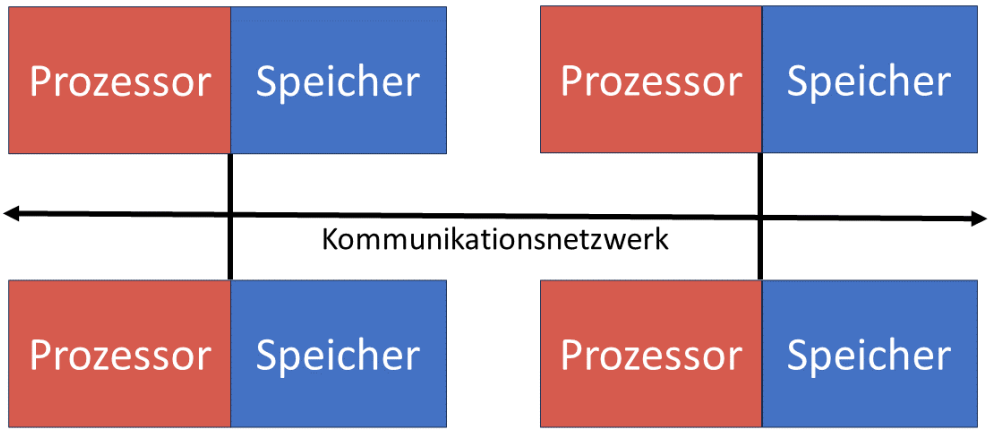


Abb. 3.5 Schematische Darstellung der Architektur mit verteiltem Speicher (engl. „distributed memory“)

Prozessor-Speichereinheiten sind über ein Kommunikationsnetzwerk miteinander verknüpft. Bei den Prozessoren handelt es sich um Ein-Kern-Prozessoren /BAR 17/.

Der größte Vorteil dieser Architektur besteht in der guten Skalierbarkeit. Prozessoren und Speicher wachsen proportional zueinander. Allerdings kann nachteilig ausgelegt werden, dass die Kommunikation zwischen den Prozessoren vom Programmierer zu erledigen ist. Zudem müssen Datenstrukturen, die auf einem globalen Adressraum beruhen, umorganisiert werden. Ein weiterer Nachteil sind die ungleichen Speicherzugriffszeiten, wenn Informationen auf einer anderen Prozessor-Speichereinheit liegen /BAR 17/. Um solche verteilten Speichersysteme zu verwenden, ist die Nutzung von „message passing“ Programmiermodellen nötig /TRO 18/, wie z. B. MPI (siehe Abschnitt 4.2).

Parallel zu den Entwicklungen der Multicore-Prozessoren und den GPUs wurde auch das „distributed memory“ System erweitert, so dass hybride Lösungen entstanden. Schematische Darstellungen solcher Systeme sind in der Abb. 3.6 und Abb. 3.7 dargestellt. Die hybride „distributed-shared memory“ Architektur wird von den größten und schnellsten Computern der Welt (engl. „High Performance Computer“ (HPC)) angewandt. Dabei können die „shared memory“ Einheiten entweder „shared memory“ Maschinen wie Multicore-Prozessoren oder GPUs sein (vgl. Abb. 3.6 und Abb. 3.7) /BAR 17/.

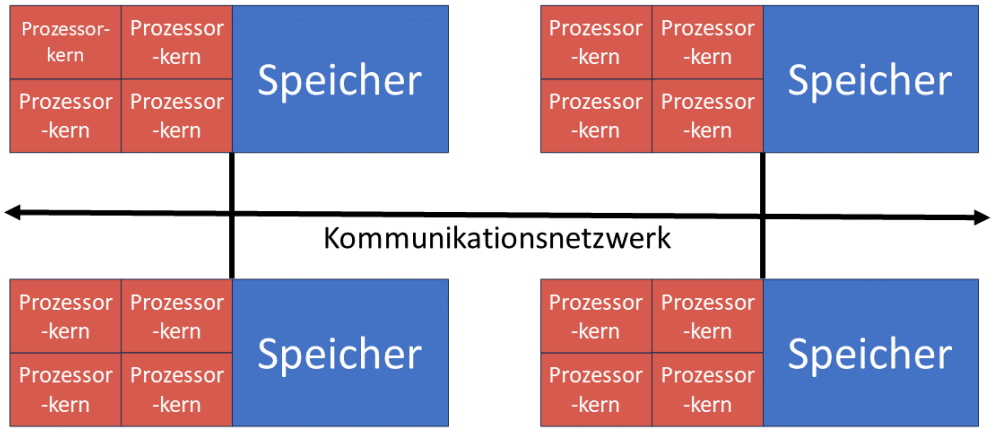


Abb. 3.6 Schematische Darstellung der hybriden Architektur mit verteiltem Speicher („distributed-shared memory“) für Multicore-Prozessoren

Prozessor-Speichereinheiten sind über ein Kommunikationsnetzwerk miteinander verknüpft. Bei den Prozessoren (rot) handelt es sich in diesem Beispiel um Vier-Kern-Prozessoren /BAR 17/.

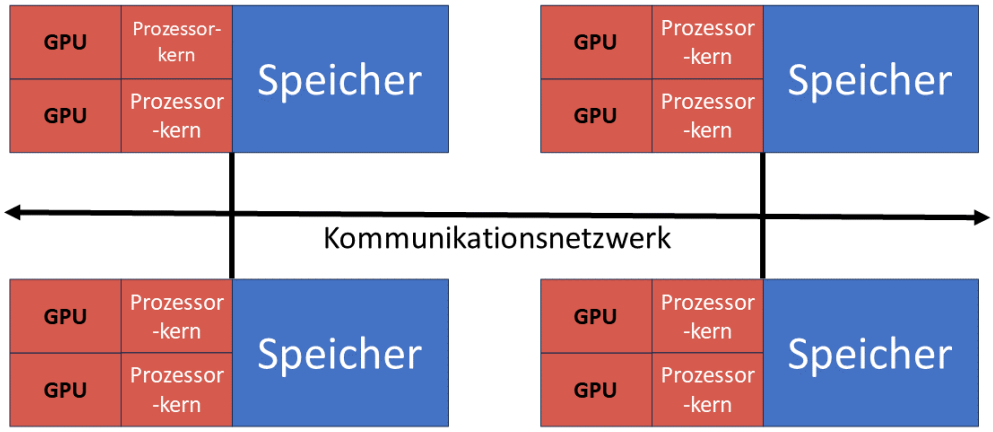


Abb. 3.7 Schematische Darstellung der hybriden Architektur mit verteiltem Speicher („distributed-shared memory“) für Multicore-Prozessoren und GPUs

Prozessor-Speichereinheiten sind über ein Kommunikationsnetzwerk miteinander verknüpft. Bei den Prozessoren (rot) handelt es sich in diesem Beispiel sowohl um Zwei-Kern-Prozessoren als auch um angeschlossene GPUs. Nach /BAR 17/.

Der besondere Vorteil dieser hybriden Lösung liegt in der Skalierbarkeit der Prozessoren. Der Aurora HPC des Argonne National Laboratory in den USA, der als zweitschnellster HPC gelistet wird (Stand: 20. November 2024), verfügt über 9.264.128 CPU- und GPU-Prozessorkerne /TOP 24/. Ein Nachteil ist die hohe Programmierkomplexität solcher Systeme /BAR 17/.

Je nach genauer Architektur des hybriden Systems müssen „message passing“, threadbasierte und/oder streambasierte Programmiermodelle eingesetzt werden /TRO 18/.

3.5 Parallelisierung in ARTM und ähnlichen Lagrange-Partikelmodellen

Im folgenden Abschnitt wurde recherchiert, inwieweit andere Lagrange-Partikelmodelle Parallelisierungsmöglichkeiten umsetzen. Die dabei verwendeten parallelen Programmiermodelle sind im Kapitel 4 im Rahmen des Arbeitspaket 2 erläutert.

3.5.1 ARTM

In ARTM wird der Begriff des verteilten Rechnens in einem anderen Kontext als in Abschnitt 3.4 verwendet, indem Teilaufgaben generiert werden, die von mehreren aufgerufenen Instanzen[4] von ARTM unabhängig voneinander parallel abgearbeitet werden /GRS 20/. Die Parallelisierung erfolgt somit nicht auf Programmebene (ARTM), sondern auf Betriebssystemebene. Eine Kommunikation zwischen den einzelnen aufgerufen Instanzen ist somit nicht möglich, wodurch es sich von dem o. g. verteilten Rechnen klar abgrenzt.

Bei den Teilaufgaben handelt es sich um die Berechnung der Wind- und Turbulenzfelder sowie die Berechnung der Konzentrations-/Depositionsfelder und Felder der statistischen Unsicherheit. Die Methodik ist lediglich dann effektiv anwendbar, wenn die Teilaufgaben voneinander unabhängig sind. Die Erstellung der Windfeldbibliothek erfolgt für jede Ausbreitungsklasse unabhängig voneinander und kann somit auf mehrere Instanzen von ARTM verteilt werden. Die Ausbreitungsrechnung selbst wird in einzelne Sequenzen der meteorologischen Eingangsdaten aufgeteilt. Je nach Ausbreitungssituation und Größe des Rechengebietes können Partikel, die innerhalb einer Sequenz emittiert wurden, am Ende der Sequenz noch im Rechengebiet liegen. Da zwischen den einzelnen Sequenzen keine Kommunikation möglich ist, werden diese Partikel nicht weiter betrachtet und verfallen in weiteren Berechnungen. Das führt zu einer Unterschätzung der Luftaktivität, die sich bei einer Simulation von mehreren Tagen, Wochen oder Monaten aber innerhalb der Simulationsunsicherheiten befinden sollte. Falls am Ende einer Sequenz alle numerischen Partikel das Rechengebiet verlassen haben, dann sind die

[4] Eine Instanz (im o. g. Zusammenhang „Anwendungsinstanz“) ist definiert als separater, abgeschlossener Programmablauf einer Anwendung mit einer eigenen, privaten Datengrundlage, die die strukturellen Voraussetzungen der Anwendung erfüllt /SHE 13/.

einzelnen Sequenzen voneinander unabhängig, und ein verteiltes Rechnen ergibt Ergebnisse, die sich nur durch statistische Variationen von der seriellen Abarbeitung der einzelnen meteorologischen Sequenzen unterscheiden.

3.5.2 MPTRAC

MPTRAC (engl. „**M**assive-**P**arallel **Tra**jectory **C**alculations“) beinhaltet ein Lagrange-Partikelmodell für die Analyse der atmosphärischen Dispersion innerhalb der freien Troposphäre und der Stratosphäre /HOF 16/. MPTRAC ist genau wie ARTM in der Programmiersprache C geschrieben und eine relativ neue Entwicklung des Jülich Supercomputing Centre aus dem Jahr 2015. Das Programm ist OpenSource und unterliegt der GNU GPL Lizenz. Seit Beginn des Projekts durchlief das Programm mehrere Optimierungsphasen und umfasst aktuell 18 Versionen mit der aktuellen Programmversion 2.7. Ein Überblick über alle Versionen von MPTRAC ist auf der Webseite von Zenodo zu finden /ZEN 24/.

Die Parallelisierung innerhalb von MPTRAC basiert auf einen Hybridansatz aus CPU-Parallelisierung, verteilten Rechnen sowie GPU-Parallelisierung und ermöglicht somit abhängig von der Hardwarearchitektur einen effizienten Einsatz der unterschiedlichen Parallelisierungsmöglichkeiten. Auf einem Cluster kann z. B. Open Multi Processing (OpenMP) die CPU-Parallelisierung der einzelnen Prozesse auf einem Knoten umsetzen, während das verteilte Rechnen durch Message Passing Interface (MPI) die Kommunikation unter den unterschiedlichen Knoten sicherstellt (verteiltes Rechnen). In Kombination mit MPI können mit OpenACC seit Version 2.0 auch Multi-GPU-Simulationen durchgeführt werden /HOF 22/. Die Umsetzung im C Code erfolgt dabei über OpenMP und OpenACC Präprozessor Direktiven sowie entsprechenden Funktionen von MPI. Präprozessor Direktiven sind dabei im Quellcode eingefügte Anweisungen (Direktiven) für den Kompilierer.

Die Parallelisierung mittels OpenMP ist standardmäßig in allen Versionen von MPTRAC aktiv. Sie wird verwendet, um den advektiven Transport der Partikel auf die Prozessorkerne eines einzelnen Rechenknotens zu verteilen. Da OpenMP auf dem Prinzip des „shared memory“ basiert, wird für die kompletten Berechnungen nur eine einzige Kopie der teils sehr großen meteorologischen Datensätze im Speicher gehalten. Typischerweise nehmen die Berechnungen der Trajektorien den größten Teil der Gesamtlaufzeit der Simulationen in Anspruch und können unabhängig voneinander durchgeführt werden. Aus diesem Grund lässt sich diese Parallelisierung sehr effektiv über einen großen

Bereich von Prozessorkernen skalieren. Bei der Parallelisierung mittels OpenMP in MPTRAC zeigte sich, dass die Performance grob linear mit der Anzahl der zur Verfügung stehenden CPUs skaliert /HOF 16/.

In MPTRAC Version 2.0 wurden OpenACC Präprozessor Direktiven verwendet, um die Berechnungen für die rechenintensiven Teile des advektiven Transports der Partikel auf die Rechenelemente der GPU zu verteilen. Weitere Präprozessor Direktiven sorgen für den relevanten Datentransfer zwischen dem Hauptspeicher und dem GPU-Speicher.

In Version 2.6 wurde eine weitere Optimierung der GPU-Parallelisierung des Kernels[5] für die Advektion der Partikel durchgeführt. Das betraf insbesondere den Speicherzugriff sowie die Datenstrukturierung der meteorologischen Eingangsdaten und die Eigenschaften der Partikel /HOF 24/. Für viele Teile des Codes, insbesondere für den Kernel der Advektion, wurde die Rechenzeit maßgeblich durch die Menge an freiem Speicher, die zum Halten der Arbeitsdaten erforderlich ist (engl. „memory-bound“), beschränkt. Ebenso erfolgte der Speicherzugriff größtenteils in beliebiger Reihenfolge (engl. „random access memory pattern“). Zufällige Speicherzugriffsmuster sind dabei im Allgemeinen schädlich für die Cache- und Speicherzugriffszeiten.

Der erste Aspekt, „memory-bound“, wurde dadurch optimiert, dass die Datenstrukturierung der meteorologischen Daten angepasst wurde. Dabei wurden die dreidimensionalen horizontalen und vertikalen Windgeschwindigkeitskomponenten (*u, v, w*), die zuvor jeweils durch einem 3D-Array, bestehend aus Float Variablen, umgesetzt wurden (z. B. `u[ex][ey][ez]`[6]), durch ein 4D-Array ausgetauscht, welches als vierte Komponente die einzelnen Komponenten (*u, v, w*) enthält (d. h. `uvw[ex][ey][ez][3]`). Technisch gesprochen entspricht dies einem Wechsel von „Structure of Arrays“ (SoAS) zu einem „Array of Structures“ (AoSs). Hintergrund dieser Änderung ist, dass in der Programmiersprache C Arrays im Speicher zeilenweise angeordnet sind. Im Fall der 3D-Arrays mit der x-Koordinate als erste Dimension, der y-Koordinate als zweite und der z-Koordinate als dritte sind die Elemente der Windgeschwindigkeitskomponenten entlang der z-Achse am nächsten zueinander im Speicher abgelegt, gefolgt von der y-Dimension und der x-Dimension. In der Umsetzung SoAS kann somit das Vertikalprofil effektiv vom Haupt-

[5] Eine Funktion, die parallel auf einer angeschlossenen GPU ausgeführt werden soll, wird als Kernel bezeichnet.

[6] `ex`, `ey`, und `ez` sind dabei die Dimensionen des Rechengitters in x-, y- und z-Richtung.

speicher auf den GPU-Speicher durch sogenannte Cache-Lines transferiert werden, d. h. der Zugriff vom Cache-Speicher zum Hauptspeicher erfolgt in einem einzigen, blockweisen Transfer. Falls jedoch der Kernel primär auf horizontale Profile zurückgreifen muss, wie es bei der Advektion größtenteils der Fall ist, ist die SoAS wenig effizient, da in der horizontalen Ebene die Daten weiter im Speicher voneinander getrennt sind als in der vertikalen Ebene und somit bei dem Transfer über Cache-Lines, die bei der GPU-Parallelisierung notwendig sind, teilweise auch unnötige Daten transferiert werden, auf die gar nicht zugegriffen werden muss. Dieses Problem wurde durch die Umstellung auf AoSs in MPTRAC gelöst.

Ein weiterer Faktor ist das Problem des zufälligen Speicherzugriffsmuster der Partikeldaten. Zu diesem Zweck wurde ein Sortierungsalgorithmus eingeführt, der es ermöglicht die Daten für die Partikeleigenschaften im Speicher besser zum allgemeinen Speicherzugriffsmuster der meteorologischen Daten zu orientieren. Der Algorithmus bewirkt dabei, dass ein lineares Iterieren über die Teilchen ermöglicht wird und dabei in einer optimierten Speicherzugriffsreihe auf die meteorologischen Daten zugegriffen werden kann. Zusammenfassend wurde also die Speicherstruktur der Partikel auf die Speicherstruktur der meteorologischen Daten angepasst.

MPTRAC ist ein sehr guter Kandidat, um Vergleiche zu ARTM zu ziehen. Zu den Vorteilen gehört u. a., dass beide Programme in C geschrieben und OpenSource sind. Als C Kompilierer wird bei MPTRAC die GNU Compiler Collection (GCC) verwendet, welcher ebenfalls für ARTM verwendet werden kann. Die Array-Struktur bei ARTM besteht zu großen Teilen aus AoSs, jedoch mit einem relativ komplizierten Table-Manager. Eine zentrale Aussage, die aus der Dokumentation von MPTRAC mitzunehmen ist, ist die genaue Speicherstruktur des Table-Managers zu studieren und im Fall auch anzupassen. Dieser Aspekt spielt jedoch primär dann eine Rolle, wenn eine Parallelisierung über die Grafikkarte realisiert werden soll.

Ein weiterer wichtiger Punkt, der aus der Dokumentation entnommen werden konnte, ist die Rechenzeit und deren Skalierbarkeit mit der Anzahl an verwendeten CPUs. In MPTRAC besteht sowohl die CPU-Parallelisierung der Advektion über OpenMP sowie die GPU-Parallelisierung mittels OpenACC.

3.5.3 FLEXPART

FLEXPART (engl. „**FLEX**ible **PART**icle Dispersion Model") ist ein Lagrange-Transport- und Dispersionsmodell, das für die Simulation einer Vielzahl atmosphärischer Transportprozesse geeignet ist. Geschrieben ist FLEXPART in der Programmiersprache Fortran 90 und ist wie ARTM und MPTRAC Open Source.

In FLEXPART wurde in der Version 10.4 zunächst die Parallelisierung durch MPI umgesetzt /PIS 19/. In Version 11 wurde diese Art der Parallelisierung durch OpenMP ersetzt /BAK 24/. Der Hintergrund der Umstellung war der Speicherzugriff. Während bei einer Parallelisierung mittels OpenMP zwischen den einzelnen Prozessen der Speicher, z. B. für die meteorologischen Eingangsfelder, gemeinsam genutzt wird („shared memory"), wird bei einer Parallelisierung mittels MPI für jede Prozessor-Speichereinheit eine separate Datenkopie benötigt. Ein Testfall zeigte, dass eine Rechnung mit 10^6 freigesetzten Simulationspartikeln mittels einer OpenMP Parallelisierung, unabhängig von der Anzahl an Prozessorkernen, in etwa 11 GB an Speicher belegt. Eine analoge Rechnung mittels MPI und 32 Prozessor-Speichereinheiten benötigte hingegen 132 GB. Ebenso wurde argumentiert, dass durch das Aufsplitten der Ausbreitungsrechnung in mehrere einzelne Simulationen, wie es auch in ARTM aktuell durch das Aufsplitten in mehrere Instanzen von ARTM möglich ist, weiterhin ein Ansatz zur Parallelisierung auf verteilte Speicher Systemen verfügbar ist.

In FLEXPART Version 11 wurden große Teile des Codes parallelisiert. Dazu gehört die Parallelisierung aller partikelabhängigen Berechnungen (ausgenommen der initialen Freisetzung), der Advektion, der nassen und trockenen Deposition sowie der vertikalen Koordinatentransformation der Felder. Testrechnungen zeigten dabei, dass bei 10^6 und 10^7 Simulationspartikeln die Rechenzeit lediglich um einen Faktor 1.3 langsamer ist als die bei einer perfekten Skalierung[7]. Hingegen steigt bei einer geringeren Anzahl von Simulationspartikeln, beispielsweise 10^5 Simulationspartikeln, der Faktor schon auf 2. Zu erklären ist dies damit, dass der Geschwindigkeitsvorteil durch die Parallelisierung den Geschwindigkeitsnachteil durch die aufwendigere Speicherverwaltung nicht kompensie-

[7] Eine perfekte Skalierung bedeutet, dass eine Reduzierung der Rechenzeit linear einhergeht mit der Anzahl an verwendeten Prozessoren, d. h. beispielsweise, wenn eine serielle Rechnung sechs Stunden dauert, dauert die identische Rechnung mittels einer Parallelisierung durch sechs Prozessoren lediglich eine Stunde. Ein Faktor von 1 bedeutet perfekte Skalierung. Faktoren größer als 1 deuten an, dass die Parallelisierung nicht perfekt ist. Serielle und parallele Ausführung wären zeitlich identisch, wenn der Faktor der Anzahl der verwendeten Prozessoren entspräche.

ren kann. Ähnliche Erkenntnisse ergaben sich auch bei der Parallelisierung mittels MPI, nämlich dass die Parallelisierung mit steigender Anzahl von Simulationspartikeln zielführender ist.

Ebenfalls in Entwicklung ist eine für die CTBTO[8] angepasste FLEXPART Version, die es ermöglichen soll, GPUs effizient zur Parallelisierung nutzen zu können /MOR 23/. Die Anpassung wird an der Version v10 vorgenommen und soll die GPU-Parallelisierung mittels des Fortran-Kompilierer nvfortran erreichen. nvfortran ist dabei kompatibel mit NVIDIA-GPUs und umfasst eine CUDA-Toolchain[9], Bibliotheken und Header-Dateien für GPU-Computing /NVI 24b/. Die Erstellung von Leistungsprofilen des Codes mit benutzerdefinierten und von NVIDIA bereitgestellten Tools zeigte, dass insbesondere der Transport der Partikel mittels der GPUs am zielführendsten zu parallelisieren sei. Im Vortrag von Herrn Morton auf der CTBT: Science and Technology Conference 2023 /MOR 23/ wird jedoch hervorgehoben, dass die einzelnen Iterationen des Partikeltransportes stark voneinander abhängig sind. Analog zu ARTM weist der FLEXPART Code eine tiefe Hierarchie von Unterprogramm- und Funktionsaufrufen auf, die zudem mit über mehr als 200 globalen Variablen stark voneinander abhängen und in vielen Fortran-Modulen verteilt und definiert sind. Aus diesem Grund wurden Teile des Codes in einfachere, schnell auszuführende und eigenständige Testprogramme extrahiert. Die Parallelisierung wurde dabei mittels OpenACC und mehr als 200 Direktiven umgesetzt. Deutlich wird im Vortrag hervorgehoben, dass die GPU-Parallelisierung von FLEXPART mittels OpenACC eine große Herausforderung darstellt. Ein Zitat aus dem Vortrag ist:

“The application of OpenACC directives – **just to get it to compile and execute** – has been quite complicated and tedious, requiring many directives, and the creation of a spreadsheet to maintain an inventory of all 200+ global module variables referenced in the underlying code.”

---

[8] Die Comprehensive Nuclear-Test-Ban Treaty Organization (CTBTO) ist eine internationale Organisation, die mit Inkrafttreten des Kernwaffenteststopp-Vertrages (CTBT) ihre Arbeit aufnehmen und sodann die Einhaltung dieses Vertrages überwachen soll.

[9] Eine Toolchain ist eine Sammlung von Softwareentwicklungstools, welche zur Erzeugung einer Software verwendet werden. Der Begriff „chain“ bedeutet dabei, dass die Tools nacheinander ausgeführt werden und von den Ergebnissen der jeweiligen Tools abhängen.

3.5.4 PALM

PALM (engl. „**PA**rallelized **L**arge-Eddy Simulation **M**odel“) wurde vom Institut für Meteorologie und Klimatologie der Leibniz Universität Hannover entwickelt und basiert auf der Programmiersprache Fortran 95 mit einigen Erweiterungen in Fortran 2003. PALM ist frei verfügbar und unterliegt der GNU GPL Lizenz.

Wie der Name schon sagt, werden Large-Eddy Simulationen (LES) für verschiedene atmosphärische und ozeanische Grenzschichten mit PALM durchgeführt /MAR 15/. Ein Lagrange-Partikelmodell ist in PALM implementiert und nimmt als Eingabe die Wind- und Turbulenzfelder der LES von PALM.

PALM ist optimiert für die Simulation auf HPCs und wird seit Version 3.9 aus dem Jahr 2013 sukzessiv für die Anwendung auf GPUs erweitert. Die Umsetzung der GPU-Parallelisierung innerhalb des Codes wird mittels OpenACC realisiert. Eine vollständige GPU-Umsetzung wurde im Jahr 2016 abgeschlossen /KNO 16/.

PALM enthält zwei Methoden der Parallelisierung, eine basierend auf einer zwei- dimensionalen Domain Decomposition Methode (DDM), eine weitere auf Schleifenebene.

Details zur DDM können /RAA 01/ entnommen werden. Kurz zusammengefasst wird das gesamte Rechengebiet in Teilbereiche (engl. „Subdomain“) unterteilt, die den Prozessoreinheiten (PE) zugeordnet werden. Pro PE gibt es einen dazugehörigen Teilbereich und jede PE löst die gesamte Menge an Modellgleichungen auf seinem Teilbereich. Die Kommunikation zwischen den PEs wird durch MPI realisiert, und Randbedingungen der zu lösenden Modellgleichungen werden implizit durch eine virtuellen zweidimensionalen Prozessortopologie realisiert, in der die Prozessoren an den entsprechenden Enden einzelner Zeilen bzw. Spalten gekoppelt sind.

Die Parallelisierung auf Schleifenebene, auf Basis der „shared memory“ Architektur, wird durch OpenMP realisiert. Im sogenannten Hybridmodus können durch verteiltes Rechnen mit MPI mehrere MPI-Prozesse mit mehreren OpenMP-Threads auf unterschiedlichen Knoten gestartet werden.

3.5.5 HYSPLIT

Im **H**ybrid **S**ingle-**P**article **L**agrangeian **I**ntegrated **T**rajectory Model (HYSPLIT) wurde eine Parallelisierung durch die „shared memory“ Architektur und OpenMP sowie durch

eine GPU-Parallelisierung auf einer NVIDIA GTX960 und Tesla P100 Grafikkarte mittels CUDA realisiert /YU 19/. Ebenso existiert eine Möglichkeit der Parallelisierung mittels MPI, welche jedoch nur effektiv für Programme mit grober Granularität ist, bei denen die Prozesse eine lange Ausführungszeit haben und damit nur wenig Datenaustausch zur Synchronisation benötigen.

Das analysierte Leistungsprofil des nicht parallelisierten Ursprungscode zeigte, dass mehr als 80 % der Rechenzeit auf Interpolationsschritte, der Dispersion sowie der Advektion innerhalb der Schleifenebene der Simulationspartikel fallen. Die Schleifenebene der Simulationspartikel besaß in ihrer Ursprungsform schon eine nahezu perfekte Parallelisierbarkeit und hatte nur zwei kritische Funktionen, die eine Parallelisierung verhinderten. Die erste Funktion war die Zuordnung eines Simulationspartikels in ein Teilrechengebiet basierend auf seiner vorherigen Iteration, die zweite Funktion die am Ende einer Schleife zu berechnende Konzentration. Um die Datenabhängigkeit dieser beiden Funktionen zu umgehen, wurde die Schleifenebene in eine serielle Schleife (Neuladen des Teilrechengebiets und weitere Datei-Eingabe/Ausgabe-Vorgänge), eine parallele Schleife (Advektion, Interpolation, Dispersion und Deposition), gefolgt von einer seriellen Schleife (zu berechnende Konzentration) aufgeteilt. Die parallele Schleife wurde dabei mittels OpenMP realisiert.

Die GPU-Parallelisierung erfolgt durch die Abarbeitung von zwei Kernelfunktionen auf der GPU, eine für Advektion und Interpolation und eine weitere für die Dispersion, mit anschließendem Übertrag der Ergebnisse von der GPU auf den Wirt.

Eine mögliche weitere Optimierung (engl. „Coarse Grain GPU Parallelization Approach“) ist eine Überlappung verschiedener Kernel-Aufrufe, welche den Speicheraufwand des Wirt-Geräts reduziert und eine schlechte GPU-Auslastung korrigiert. Dabei gibt es in HYSPLIT drei verschiedene Ansätze:

- Single-Thread-Ansatz: Ein einzelner CPU-Thread weist gemäß des Round-Robin-Verfahrens[10] Aufgaben den verschiedenen CUDA-Streams[11] zu.

[10] Ein Verfahren, bei dem die Aufgaben reihum und gleichmäßig an die abarbeitende Struktur verteilt werden.

[11] Ein Stream ist eine Reihe von Befehlen, die der Reihe nach ausgeführt werden. In CUDA-Anwendungen erfolgen die Kernel-Ausführung sowie einige Speicherübertragungen innerhalb von CUDA-Streams.

- Multi-Thread-Ansatz: Jeder CPU-Thread weist Aufgaben seinem eignen CUDA-Stream zu.
- Multiprozess-Ansatz: Jeder MPI-Prozess weist Aufgaben seinem eignen CUDA-Stream zu.

Für die Parallelisierung mittels OpenMP wurde bei Performancetests eine nahezu perfekte Skalierung mit minimalem seriellem Overhead erreicht. Bei der GPU-Parallelisierung ergab sich eine 4 – 5-mal schnellere Gesamtleistung als das Original ohne Parallelisierung ausgeführt auf einer CPU. Mit Hilfe der Überlappung verschiedener Kernel-Aufrufe wurde eine maximale Beschleunigung um das 12,9-fache im Vergleich zum Originalprogramm erreicht.

3.5.6 Parallel Micro-SWIFT-SPRAY (PMSS)

Parallel **M**icro-**S**WIFT-**S**PRAY (PMSS) ist die parallele Programmierungsumsetzung der Kombination aus dem Windfeldmodell Micro-SWIFT und dem Lagrange-Partikelmodell Micro-SPRAY. Die Parallelisierung in PMSS wird durch verteiltes Rechnen mit MPI realisiert /OLD 11/.

SWIFT wird über zwei Wege parallelisiert, einerseits der DDM ähnlich zu PALM, und andererseits über die zeitliche Parallelisierung (engl. „time frame parallelization“ (TP)). Beide Wege können parallel genutzt werden. Bei der DDM wird das Rechengitter in kleinere Teilbereiche unterteilt, so dass sie in den Speicher eines einzelnen Prozessorkerns passen. Liegen nach der initialisierten DDM noch Prozessorkerne für die TP vor, werden parallel verschiedene Zeitschritte auf den Prozessorkernen berechnet. Da zwischen zwei einzelnen Zeitschritten keine Kommunikation erforderlich ist, ist die Beschleunigung unter TP sehr effizient. Bei der DDM erfordert jeder einzelne Schritt des Lösungsalgorithmus eine MPI-Kommunikation zwischen benachbarten Teilbereichen, sowohl für den massenkonsistenten als auch für den RANS-Löser in SWIFT. Neben den Prozessorkernen, die die Parallelisierung und Rechnungen durchführen, braucht SWIFT auch einen sogenannten Master Prozessorkern (engl. „master core“), welcher die Aufgaben und Eingangsdaten auf die entsprechenden Prozessorkerne verteilt.

Die Parallelisierung von SPRAY baut auf der übergebenen Datenstruktur von SWIFT auf. Falls SWIFT lediglich ein Rechengitter übergibt, wird SPRAY im sogenannten Single-Tile Parallelization (STP)-Modus betrieben. Der STP-Modus benötigt nur eine geringe Kommunikation zwischen den einzelnen Prozessorkernen und verteilt mittels MPI

die emittierten Simulationspartikel auf die zur Verfügung gestellten Prozessorkerne. Da die emittierten Simulationspartikel zufällig den Prozessorkernen zugeteilt werden, ergibt sich ein Gleichgewicht zwischen schnellen und langsam dispergierenden Simulationspartikeln, so dass jeder Prozessorkern ähnlich lange für die Berechnung braucht. Ebenso wird jeder Prozessorkern mit einer anderen Anfangszahl des Zufallszahlengenerators gestartet, um die statistische Eigenschaft des Modells aufrecht zu halten. Zum Zeitpunkt der Synchronisierung, d. h. am Ende jedes Zeitschrittes, summiert der Master Prozessor die dreidimensionalen Konzentrationsfelder der einzelnen Prozessorkerne auf. Diese werden dem Master Prozessor über MPI-Kommunikation von den einzelnen Prozessorkernen zur Verfügung gestellt.

Falls SWIFT mehrere separate Teilgebiete des Rechengitters an SPRAY übergibt, aktiviert SPRAY den Multiple-Tiles Parallelization (MTP)-Modus. In einem ersten Schritt werden beim Aufruf der Routinen für den allerersten Zeitschritt alle Teilgebiete aktiviert, in denen eine Quelle vorliegt. Alle anderen werden deaktiviert. Abhängig von den Massenraten der vorliegenden Quellen, werden anschließend die zur Verfügung stehenden Prozessorkerne auf die aktivierten Teilgebiete verteilt. Liegen beispielsweise zwei Quellen an unterschiedlichen Stellen im Rechengebiet vor und besitzt die eine Quelle eine doppelte Massenrate als die andere, werden dem Teilgebiet mit der doppelten Masse entsprechend auch doppelt so viele Prozessorkerne mittels spezifischer MPI-Kommunikatoren (engl. „specific MPI communicators“ (SMPIC)) zugeteilt. Im nächsten Schritt werden den Prozessorkernen in den Teilgebieten die Simulationspartikel zugeteilt. Dieses erfolgt analog zum STP-Modus. Da Simulationspartikel dispergieren und somit Teilgebiete verlassen oder aus anderen Teilgebieten einfließen können, sind die einzelnen Teilgebiete nicht unabhängig voneinander. Aus diesem Grund muss jedes Teilgebiet einen Master Prozessorkern besitzen, der zum Zeitpunkt der Synchronisation sowohl Simulationspartikel an andere Teilgebiete sendet als auch von anderen Teilgebieten empfängt. Falls zum Zeitpunkt der Synchronisation Simulationspartikel in ein noch nicht aktiviertes Teilgebiet dispergiert sind, werden die Prozessorkerne anderer Teilgebiete für das nun aktivierte Teilgebiet zur Verfügung gestellt. Da durch das Dispergieren der Simulationspartikel schnell ein Ungleichgewicht zwischen den aktivierten Teilgebieten und den darin enthaltenen Simulationspartikeln entstehen kann, wurde in SPRAY ein sogenannter „Load Balancing Process“ entwickelt. Dieser berechnet zunächst durch den Master Prozessorkern in jedem Teilgebiet die Gesamtanzahl an Simulationspartikeln. Je mehr Simulationspartikel in einem Teilgebiet vorliegen, desto mehr verfügbare Prozessorkerne werden diesem Teilgebiet zugeteilt. Die dazu notwendigen Prozessorkerne werden dabei von Teilgebieten weggenommen, die im Vergleich eine geringe Anzahl an

Simulationspartikeln aufweisen. In einem letzten Schritt werden erneut die Simulationspartikel auf die für das Teilgebiet zugeteilten Prozessorkerne verteilt.

Für ARTM ist MTP wahrscheinlich nicht zielführend, da der „Load Balancing Process“ am Ende jedes Zeitschrittes viel Rechenzeit benötigt. MTP wird insbesondere dann interessant, wenn das Rechengebiet sehr groß ist. Die PMSS Simulation im Rahmen des EMED Projektes umfasste beispielsweise ein Rechengebiet von 19.333 x 16.666 x 39 Gitterzellen mit einer horizontalen Auflösung von 3 m und einer vertikalen Auflösung von 1 m in Bodennähe /OLD 19/. Die Leistungsfähigkeit von MTP wurde für einzelne Teilgebiete mit der kleinsten Auflösung von 201 x 201 Punkten bis hin zu einer Auflösung von 1001x1001 getestet. Dadurch ergaben sich 8051 bzw. 340 Teilgebiete. Zur Verfügung standen 500 bis 3.000 Prozessorkerne. Anhand dieser Zahlen ist deutlich zu erkennen, dass MTP für ARTM nicht geeignet ist, da tendenziell schon die kleinsten Teilgebiete in der Größenordnung der maximal erlaubten Gitterzellen in der horizontalen Ebene von ARTM liegen. Für ARTM ist somit lediglich die STP von Relevanz. Tests der Leistungsfähigkeit zeigten dabei eine nahezu perfekte Skalierung. Details zur Dimension des Rechengitters sind jedoch nicht angegeben, so dass eine Bewertung hinsichtlich der Effektivität der STP für ARTM nicht möglich ist.

3.5.7 Zusammenfassung

Die Tab. 3.1 gibt einen Überblick über die verwendeten Parallelisierungsmodelle innerhalb der betrachteten Lagrange-Partikelmodelle.

Tab. 3.1 Überblick über die verwendeten Parallelisierungsmodelle innerhalb der betrachteten Lagrange-Partikelmodelle

Modell	OpenMP	MPI	OpenACC	CUDA
MPTRAC	✓	✓	✓	×
FLEXPART	✓	×[1]	(✓)[2]	×
PALM	✓	✓	✓	×
HYSPLIT	✓	✓	×	✓
PMSS	×	✓	×	×

[1] Aktuelle Version nur noch OpenMP, zuvor Parallelisierung mittels MPI
[2] Anpassung an Version 10.0 für die CTBTO, noch in Entwicklung

Nahezu alle Modelle enthalten eine Parallelisierung mittels der „shared memory“ Architektur und dem Standard OpenMP. Alle Modelle haben dabei den Transport der Simulationspartikel als kritischsten Punkt der Laufzeit identifiziert und somit diesen parallelisiert.

In mehreren Publikationen wird hervorgehoben, dass das verteilte Rechnen mittels MPI nur dann sinnvoll ist, wenn für die Rechnungen eine grobe Granularität gegeben ist, d. h. die Prozesse besitzen eine lange Ausführungszeit bei nur wenig Datenaustausch zur Synchronisation. Dies ist insbesondere für hochaufgelöste und große Rechengebiete der Fall, wie z. B. das PMSS Rechengebiet von 19.333 x 16.666 x 39 Gitterzellen mit einer horizontalen Auflösung von 3 m und einer vertikalen Auflösung von 1 m in Bodennähe innerhalb des EMED Projektes /OLD 19/. Ein weiteres Beispiel sind die Large-Eddy Simulationen in PALM, die als Grundvoraussetzung sehr feine Gitterstrukturen benötigen. Ebenso ist zu beachten, dass Modelle wie MPTRAC, PALM und auch HYSPLIT für massiv parallele Computerarchitekturen programmiert sind. Durch die Vielzahl an zur Verfügung stehenden Knoten ist dabei eine Parallelisierung mittels MPI zielführend. Die Ansprüche sind jedoch nicht unbedingt identisch zu denen von ARTM, da ARTM nicht primär für massiv parallele Computerarchitekturen ausgelegt sein soll.

FLEXPART wurde in der neuesten Version von MPI auf OpenMP umgestellt, da der notwendige Speicher durch eine Parallelisierung mittels MPI zu groß wurde und die Speicherverwaltung länger brauchte als die eigentliche Aufgabe.

Tendenziell sieht die GRS ARTM eher im Bereich der feinen Granularität, d. h. die Ausführungszeit bzw. Arbeitsweise der einzelnen Aufgaben ist klein bzw. schnell im Vergleich zur Abarbeitungszeit des kompletten Programmes. Aus diesem Grund wird für eine feinkörnige Parallelität ein Programm in eine große Anzahl kleiner Aufgaben zerlegt und einzeln vielen Prozessoren zugewiesen. Da jede Aufgabe somit weniger Daten verarbeiten muss und aufgrund dessen nur eine geringe Datenmenge zur Synchronisation notwendig ist, diese aber sehr häufig unter den Prozessen ausgetauscht werden müssen, ist solch eine Architektur am effektivsten parallelisierbar mittels der „shared memory“ Architektur und somit mittels OpenMP. Aus diesem Grund schätzt die GRS für die Schleifenebene des Partikeltransports die Umsetzung analog zu der von HYSPLIT als am sinnvollsten für ARTM ein.

Eine Parallelisierung mittels MPI in ARTM macht lediglich dann Sinn, wenn ARTM auf einem Cluster laufen soll, auf dem die unterschiedlichen Knoten parallelisiert werden sollen. Hierzu müssen aber auch genügend parallel auszuführende Aufgaben vorliegen, die unabhängig voneinander abgearbeitet werden können und eine grobe Granularität aufweisen. Hier bietet sich beispielsweise die Berechnung der Wind- und Turbulenzfelder unter dem Einfluss von Bebauung und Gelände an. Für jede der sechs Ausbreitungs-

klassen müssen in 10°-Schritten 36 Anströmungsprofile erzeugt werden. Somit ergeben sich 6 x 36 = 216 einzelne Aufgaben, die mittels MPI parallelisiert werden könnten.

Eine Parallelisierung auf der GPU wird größtenteils durch OpenACC realisiert. CUDA wird lediglich von HYSPLIT verwendet. Die Verwendung von CUDA oder auch OpenCL innerhalb von ARTM würde ein komplettes Neuschreiben des existierenden Codes verlangen. Daher tendiert die GRS dazu, eine GPU-Parallelisierung mit direktiven-basierten Standards wie OpenACC und OpenMP umzusetzen.

Diese Aspekte werden im Rahmen des Arbeitspaket 3 genauer betrachtet.

3.6 Zusammenfassung

Im Arbeitspaket 1 wurden hardwarenahe Parallelisierungsmöglichkeiten recherchiert. Da diese Parallelisierungsmöglichkeiten untrennbar mit der zugrundeliegenden Hardware verknüpft sind, wurde in einem ersten Schritt die grundlegende Von-Neuman Rechnerarchitektur eingeführt. Darauf aufbauend wurden verschiedene Architekturkonzepte vorgestellt, die eine Parallelisierung ermöglichen. Dazu zählen die Parallelisierung auf CPU- und GPU-Ebene, sowie das Prinzip des verteilten Rechnens. Maßgebliche Unterschiede dieser Parallelisierungsmöglichkeiten sind in den Speicherarchitekturen zu finden. In einem zweiten Schritt wurde recherchiert, welche Konzepte von in der Literatur genannten Lagrange Partikelmodellen angewendet werden.

Unter der CPU-Parallelisierung versteht man im Allgemeinen die parallele Abarbeitung von Teilaufgaben in mehreren Prozessorkernen, die physikalisch jedoch auf einer CPU zusammengefasst sind. Dabei greifen die Prozessorkerne auf einen gemeinsamen Speicherbereich zu. Diese Konfiguration eignet sich besonders für die Lösung von aufgabenparallelen Problemen.

Ein anderes Konzept wird bei der GPU-Parallelisierung verfolgt. Hierbei werden sehr viele Berechnungseinheiten, die sich deutlich von Prozessorkernen unterscheiden, zusammengefasst. Der Unterschied zur CPU besteht im Wesentlichen darin, dass die Berechnungseinheiten so konzipiert sind, dass sie gleichzeitig die gleichen Maschinenbefehle, aber auf unterschiedliche Datenströme, ausführen. Trotz eines gemeinsamen Speichers werden die Datenströme voneinander getrennt gehalten, damit keine Speicherzugriffsprobleme entstehen. Die große Anzahl von eher einfachen Berechnungseinheiten führt zu einem hohen Grad der parallelen Abarbeitung.

Das Konzept der Vernetzung von mehreren physikalischen Maschinen wird als verteiltes Rechnen bezeichnet. Dabei spielt es keine Rolle, ob die Maschinen nur über CPUs oder auch über GPUs verfügen. Hierbei verfügt jede Maschine über einen eigenen Speicher. Für die Verteilung der Daten auf die Speicher und den Datenaustausch ist der Programmierer zuständig.

Aus der Recherche zur Parallelisierung von Lagrange Partikelmodellen geht hervor, dass es nicht die eine Methode gibt, die allen Ansprüchen genügt. Vielmehr muss das Parallelisierungskonzept zur Arbeitsweise des Modells, den Anwendungsanforderungen und der zugrundeliegenden Hardware passen. Dennoch wurde die Tendenz ersichtlich, dass die Verwendung von Direktiven-basierten Single Source Programmiermodellen für eine breite Hardwarebasis und viele Ausbreitungsmodelle gewählt wurde, da sich diese Programmiermodelle durch eine einfachere nachträgliche Parallelisierung eines Ausbreitungsmodells auszeichnen. Dennoch sind diese nicht als Universalwerkzeug zu verstehen, da die nachträgliche Parallelisierung in allen betrachteten Fällen einen erheblichen Eingriff in die Implementierung bedingte.

4 Arbeitspaket 2: Recherchen zu möglichen Bibliotheken, Compilern, Grafikkarten und deren Anwendbarkeit für ARTM

Programmiermodelle bieten einen konzeptionellen Rahmen, der die Sichtweise des Programmierers auf den Computer definiert und festlegt, wie dieser damit interagiert, um Software zu schreiben. In diesem Kapitel wurde eine Auswahl von parallelen Programmiermodellen recherchiert, die für die Parallelisierung einer komplexen Software wie ARTM in Frage kommen könnte. Die Auswahl erfolgte zum einen nach der Eignung für Desktop-Computer und der unterstützten Programmiersprache, zum anderen aber auch nach der Verwendung der Programmiermodelle in anderen Ausbreitungsmodellen. Die gewählten Programmiermodelle werden in diesem Kapitel vorgestellt.

Zusätzlich zu diesen Recherchen nahmen zwei Mitarbeiter der GRS an jeweils zwei Schulungen teil. Die Schulungen wurden vom Institute for Advanced Simulation (IAS) am Jülich Supercomputing Centre (JSC) und dem Zentrum für Nationales Hochleistungsrechnen Erlangen (NHR@FAU) angeboten. Die Titel der Schulungen lauteten „Introduction to parallel programming with OpenMP“ (NHR@FAU) und „Directive-based GPU programming with OpenACC“ (JSC) und fanden vom 4. – 5. September 2024 bzw. vom 29. – 31. Oktober 2024 als Onlineveranstaltungen statt.

4.1 OpenMP (Open Multi Processing)

OpenMP ist ein Standard, der eine plattformunabhängige Programmierschnittstelle für „shared memory“ (geteilten Hauptspeicher) Parallelisierung definiert. Damit bietet OpenMP eine standardisierte Möglichkeit, um Anwendungen mit Threads zu parallelisieren /OPE 19/. Da dieser Standard von mehreren Kompilierer- und Hardwareherstellern ins Leben gerufen und entwickelt wurde, ist dieser einer der verbreitetsten, wenn es um die Parallelisierung von Software geht.

Durch die Entwicklung von OpenMP für „shared memory“ Architekturen lässt es sich für die parallele Programmierung auf PCs ebenso wie auf vielen HPCs verwenden. Dabei wird die Parallelisierung hauptsächlich auf Basis der vorhandenen CPUs, z. B. in einem Mehr-Kern-Prozessor, durchgeführt. Zusätzlich ist es seit OpenMP 4.5 möglich, Aufgaben an einen Hardwarebeschleuniger (engl. „Accelerator“) auszulagern /OPE 19/. Dabei

handelt es sich u. a. um Grafikkarten mit GPUs oder „field programmable gate arrays“[12] (FPGAs) /IBM 24/.

OpenMP verfügt über eine sehr breite Kompatibilität mit verschiedenen GPU-Herstellern. So werden sowohl Grafikkarten der Hersteller NVIDIA und AMD wie auch Intel unterstützt (Stand: November 2023) /HER 23/. Allerdings müssen dafür verschiedene Kompilierer verwendet werden, da die Unterstützung für Intel-Grafikkarten beispielsweise nur über die Intel-Kompilierer bereitgestellt wird. Das hat zur Folge, dass die ausführbare Datei eines Programms architektur- und hardwarespezifisch sein muss, wenn eine Grafikkarte zur Beschleunigung der Berechnungen verwendet werden soll.

Bei ARTM handelt es sich um einen Quellcode, der über viele Jahre, im Falle der Programmteile von Austal2000 über mehrere Jahrzehnte, weiterentwickelt wurde. Allerdings gibt es grundlegende Funktionalitäten, die noch auf veralteten Programmierstandards der Programmiersprache C beruhen. Neuere Kompilierer, die für OpenMP nötig sind, unterstützen diese veralteten C-Standards unter Umständen nicht mehr. Das könnte dazu führen, dass große Teile des Quellcodes aufbereitet und modernisiert werden müssten.

4.2 MPI (Message Passing Interface)

MPI ist ein Standard, der über einen Nachrichtenaustausch zwischen Prozessen ermöglicht, dass Daten zwischen parallellaufenden Prozessen ausgetauscht werden können /BAR 24/. Die Verwendung von MPI ermöglicht es somit Programmen, über die Prozessoren und den „shared memory“ eines einzelnen Rechners hinaus, auf den verteilten Speicher und die Prozessoren mehrerer vernetzter Rechner zuzugreifen, um so die verwendete Rechenleistung zu erhöhen. MPI ist daher für zu parallelisierende Aufgaben auf einem Rechencluster mittels eines Nachrichtenaustausches, z. B. über TCP (engl. „Transmission Control Protocol“), geeignet, als auch auf dedizierten Rechnern, indem eine Kommunikation zwischen der CPU mit mehreren GPUs mittels MPI realisiert werden kann (vgl. Abschnitt 3.4). Ein Anwendungsbeispiel mit kombinierten Programmiermodellen ist die Nutzung von MPI mit OpenACC (siehe Abschnitt 4.4).

[12] Integrierte Schaltungen mit einer programmierbaren Hardwarestruktur. Anders als bei anderen Bauteilen, wie GPUs, sind die Schaltungen zwischen Transistoren auf einem FPGA Chip nicht hardware-seitig eindeutig definiert, sondern können nach Bedarf neu programmiert werden /INT 25b/.

Der Standard MPI ist in gängigen Programmiersprachen, z. B. C/C++, Fortran sowie Python, implementiert. Mögliche Implementierungen sind dabei MPICH, Open MPI, Parastation MPI, MVAPICH, IBM Platform MPI und Intel MPI /BAR 24/. Open MPI und MPICH sind mit den Kompilierern GNU, Intel und Clang kompatibel, während Intel MPI nur mit Intel und GNU verwendbar ist. Open MPI ist eine nicht-profitorientierte Open-Source Implementierung von MPI, welches zusätzlich von einer großen Gemeinschaft aus Industrie und Wissenschaft entwickelt und gewartet wird. Aufgrund seiner großen Flexibilität (plattformunabhängig, kompatibel mit vielen Kompilierern und sehr guten Dokumentationen) wird Open MPI von vielen der TOP500 Supercomputern verwendet /NEW 23/.

Alle Implementierungen haben gemeinsam, dass der Austausch der Nachrichten über Programmierschnittstellen realisiert wird und sogenannte Kommunikatoren definiert werden. Kommunikatoren sind eine Gruppe von Prozessoren, die untereinander kommunizieren können. Innerhalb der Gruppe besitzt jeder Prozess einen eindeutig zugewiesenen Rang (engl. „Rank“), und die Kommunikation untereinander erfolgt über ihre Ränge. Eine Möglichkeit des Nachrichtenaustausches ist die sogenannte Punkt-zu-Punkt Kommunikation (engl. „Point-to-Point Communication“) /NEW 23/. Dabei findet eine Kommunikation zwischen zwei Prozessoren statt, bei denen einer der sendende Prozess (Sendeprozess) und einer der empfangende Prozess (Empfängerprozess) ist. Der Sendeprozess übermittelt in MPI mittels einer Sendeoperation, z. B. der Funktion `MPI_Send`, Nachrichtenpakete mit dessen Rang und eines eindeutigen Tags zur Identifizierung dieses Nachrichtenpaketes an den Empfängerprozess, die dieser mittels einer Empfangsoperation, z. B. der Funktion `MPI_Rec`, empfängt. Somit muss zu jeder Sendeoperation eine passende Empfangsoperation existieren. Eine weitere Möglichkeit ist die kollektive Kommunikation, bei der alle Prozesse durch Routinen untereinander auf verschiedene Art und Weisen kommunizieren können. Bei der kollektiven Kommunikation ist jedoch auf die Synchronität zwischen den Prozessen zu achten, d. h. alle Prozesse müssen einen Punkt im Programmcode erreichen, bevor sie erneut mit der Ausführung beginnen können. Die Sicherstellung dieser Synchronität ist über unterschiedlichste Routinen möglich, z. B. `MPI_Barrier`.

Ein Vorteil einer Parallelisierung mittels der MPI-Standards ist deren Flexibilität, d. h.

- die MPI-Standards sind mit nahezu allen HPCs kompatibel,

- eine Portierung der Anwendung auf eine andere Plattform, die den MPI-Standard unterstützt, geht mit nahezu keinen oder sogar keinen Änderungen am Quellcode einher sowie
- eine große Anzahl von Implementierungen der MPI-Standards liegt vor, sowohl von Herstellern als auch im öffentlichen Bereich.

Ebenso ist ein weiterer Vorteil die Skalierbarkeit, die in der Einleitung zu MPI genauer erklärt wurde.

Zentrale Nachteile von MPI sind einerseits die Komplexität der Implementierung und der parallele Overhead. Der parallele Overhead ist dabei die erforderliche Zeit, die für die Organisation der parallelen Aufgaben notwendig ist. Dazu gehört u. a.

- der Start von Tasks („Aufgaben“),
- die Sicherstellung der Synchronität,
- die Datenkommunikation sowie
- die Beendung von Tasks.

Bei FLEXPART hat sich beispielsweise gezeigt, dass der parallele Overhead für eine Parallelisierung ein zentraler Nachteil war. Weitere Informationen diesbezüglich sind im Abschnitt 3.5.3 zu finden.

4.3 OpenCL (Open Computing Language)

OpenCL ist ein offener, gebührenfreier und plattformübergreifender Standard für die parallele Programmierung von verschiedensten Hardwarebeschleunigern, wie sie in Supercomputern, Cloud Servern, PCs, Mobilen Geräten oder eingebetteten Bauteilen zu finden sind. Zu den unterstützten Hardwarebeschleunigern zählen u. a. CPUs, GPUs, „Digital Signal Processors“ (DSPs), FPGAs und Tensor Prozessoren. Entwickelt und veröffentlicht wird OpenCL von der Khronos Group, einem nicht-profitorientierten Konsortium aus verschiedenen Unternehmen der Tech-Industrie /OPE 23b/.

Der Aufbau von OpenCL unterscheidet sich signifikant von anderen hier behandelten parallelen Programmiermodellen. Ein Programm, das mit OpenCL parallelisiert wird, ist in zwei Teile aufgeteilt, einen Wirt-Code und einen oder mehrere sogenannter Kernels

/TEX 18/. Der Wirt-Code beinhaltet alle Befehle, um die Teilaufgaben an die verschiedenen Hardwarebeschleuniger zu verschicken. Dazu gehören Befehle zur Initialisierung von OpenCL, der Speicherverwaltung, der Partitionierung der Aufgabe und der Verteilung der Teilaufgaben. Da OpenCL ein Low-Level-Programming Framework ist, wird eine sehr hardwarenahe Programmierung ermöglicht. Allerdings bedeutet das auch einen großen Programmieraufwand für die Programmierenden /KAE 15/. Die erforderlichen Funktionalitäten von OpenCL werden für den Wirt-Code durch die Einbindung einer Bibliothek realisiert. Diese liegen in den Programmiersprachen C und C++ vor. Daher muss der Wirt-Code ebenfalls in einer dieser Sprachen geschrieben sein. Der fertige Wirt-Code wird für das entsprechende Zielrechnersystem in eine ausführbare Datei kompiliert /TEX 18/. Die Kernels beinhalten in separaten Quellcodedateien die Teilaufgaben, die auf einem bestimmten Hardwarebeschleuniger ausgeführt werden sollen. Anders als beim Wirt-Code, werden diese Kernels i. d. R. nicht für ein Zielrechnersystem kompiliert. Von den Hardwareherstellern werden OpenCL Treiber zur Verfügung gestellt, die auf dem Zielrechnersystem installiert sein müssen. Diese Treiber übernehmen zum spätmöglichsten Zeitpunkt die Kompilierung der Kernels, also bei der Ausführung des Programms /OPE 23b/. Dadurch wird eine hohe Portabilität des erstellten Programmcodes gewährleistet. Für die Implementierung der Kernels definiert OpenCL eine eigene Programmiersprache namens OpenCL C. Diese stellt einen C Dialekt dar, der auf dem C99 Standard aufbaut /OPE 23b/, /KAE 15/.

OpenCL Hardwaretreiber sind für alle gängigen Grafikkarten und Prozessoren verfügbar. Darunter zählen u. a. Bauteile von Intel, AMD, NVIDIA, Texas Instruments, Apple, ARM und Qualcomm /KHR 24/.

Um den Portabilitätsvorteil von OpenCL nutzen zu können, müssen auf jedem Zielrechnersystem die entsprechenden Hardwaretreiber installiert sein. Außerdem ist zu berücksichtigen, dass bestehende Programmcodes oft einem umfangreichen Refactoring unterzogen werden müssen, um die Berechnungsaufgaben in die von OpenCL benötigte Struktur zu bringen.

4.4 OpenACC (Open Accelerators)

OpenACC basiert auf ähnlichen Ansätzen wie OpenMP. Mittels direktiven-basierter Programmierung von Hardwarebeschleunigern, wie z. B. Grafikkarten oder FPGAs, kann eine portable und plattformunabhängige Programmierung verschiedener Hardwarebeschleuniger ermöglicht werden.

OpenACC unterstützt das Auslagern von Berechnungen und Daten von einem Wirt-Gerät auf ein Beschleuniger-Gerät. Dabei können die Geräte gleich oder völlig unterschiedliche Architekturen aufweisen, wie es z. B. der Fall bei CPU-Wirten und GPU-Beschleunigern ist. Die beiden Geräte können separate Speicher oder einen einzelnen gemeinsamen Speicher haben. Für den Fall, dass die beiden Geräte separate Speicher besitzen, analysiert der entsprechende OpenACC-Kompilierer den Code, organisiert die Speicherverwaltung des Hardwarebeschleunigers und ermöglicht die Übertragung von Daten zwischen Wirt- und Hardwarebeschleunigerspeicher.

OpenACC.org ist eine nicht-profitorientierte Organisation und wird von Mitgliedern aus Industrie und Wissenschaft unterstützt /OPE 23a/. Aus diesem Grund verfügt OpenACC über eine sehr breite Kompatibilität mit den GPU Herstellern NVIDIA und AMD. Lediglich die Unterstützung von Intel-Grafikkarten ist stark limitiert (Stand: November 2023) /HER 23/. Intel ermöglicht es jedoch mit einem Programm namens „Intel® Application Migration Tool for OpenACC* to OpenMP* API“ eine direktiven-basierte Programmierung in OpenACC in eine direktiven-basierte Programmierung in OpenMP (Version 5.0 oder höher) umzuwandeln /INT 24/.

Der von ARTM verwendete GNU Kompilierer kann die Spezifikationen und die zugehörigen Funktionen von OpenACC integrieren. Der Kompilierer unterstützt die NVIDIA PTX (nvptx) und AMD Radeon (Graphics Core Next, GCN, and Instinct, CDNA) Grafikkarten /GCC 24/. Die Parallelisierung mittels NVIDIA Grafikkarten ist ebenfalls mit dem NVIDIA HPC Software Development Kit realisierbar und umfasst zusätzliche Bibliotheken für eine optimierte Nutzung der GPUs (z. B. GPU-optimierte mathematische Bibliotheken wie cuBLAS) /NVI 24b/.

4.5 Parallele Programmierungsmodelle der Grafikkartenhersteller

Sowohl NVIDIA als auch AMD stellen für ihre Grafikkarten eigene parallele Programmiermodelle zur Verfügung. Das bekanntere, und historisch gesehen erste, ist die CUDA (Compute Unified Device Architecture) Softwaresammlung von NVIDIA. Dem gegenüber steht die ROCm Softwaresammlung von AMD. Beide werden in den folgenden Abschnitten thematisiert.

4.5.1 CUDA von NVIDIA

CUDA ist eine Softwaresammlung, die aktuell unter dem Namen CUDA-Toolkit von NVIDIA frei zur Verfügung gestellt wird. Darin sind CUDA Hardware-Treiber, mathematische Bibliotheken, Kompilierer, Debugger und Profiler enthalten, die die Entwicklung von Anwendungen für NVIDIA GPUs unterstützen. Außerdem beinhaltet die Sammlung die CUDA-API und die CUDA Laufzeitumgebung. Die CUDA-API ist die Schnittstelle, die die Befehle zur parallelen Programmierung der GPU bereitstellt. Sie stellt eine Erweiterung der Programmiersprache C/C++ dar /OH 12/.

Die CUDA-API ist eine sogenannte „Single-Source“ Programmiersprache, d. h. der Wirt-Code und der Kernel-Code werden zusammen in eine Quellcodedatei geschrieben. Anstelle der üblichen Endungen *.c oder *.cpp für C bzw. C++ Dateien erhält diese bei CUDA die Endung *.cu. Diese Datei wird dann mit dem CUDA-Kompilerer „nvcc“ kompiliert. Es entsteht eine ausführbare Datei, analog zu herkömmlichen C/C++ Programmen /HAR 17/.

CUDA ist ein hardwarenahes Programmiermodell auf einer sehr niedrigen Abstraktionsebene. Dadurch ist der Programmierer für die Speicherverwaltung, die Partitionierung von Aufgeben in Teilaufgaben, die Teilaufgabenverteilung und die Synchronisierung selbst verantwortlich /HAR 17/. Darüber hinaus funktioniert die GPU Programmierung mit CUDA nur bei NVIDIA Grafikkarten. Das CUDA-Toolkit steht für Windows und Linux Betriebssysteme zur Verfügung /NVI 24c/.

4.5.2 ROCm/HIP von AMD

ROCm von AMD ist, vergleichbar mit CUDA, ebenfalls ein Softwaresammlung, die verschiedene Treiber, mathematische und weitere Bibliotheken, Kompilierer, Debugger, Profiler und andere Entwicklungssoftware enthält /ADV 24a/. Außerdem wird eine API namens HIP (Heterogeneous-computing Interface for Portability) bereitgestellt. Dabei handelt es sich um eine ordnende (engl. „marshalling“) Programmiersprache, die auf C++ basiert /ADV 24b/. Die Open-Source Parallelisierungsplattform ROCm mit ihrer Schnittstelle HIP ist besonders für heterogene Anwendungen geeignet, bei denen sowohl CPUs wie auch GPUs verwendet werden /ADV 24a/.

HIP gehört ebenfalls zu den „Single-Source“ Programmiersprachen. Dabei wird der Quellcode, analog zu herkömmlichen C Programmen, mit einem Kompilierer zu einer

ausführbaren Datei kompiliert. Dazu muss der HIP-Kompilierer „hipcc“ verwendet werden. Alternativ besteht die Möglichkeit, die Kernel-Teile des Quellcodes erst zur Laufzeit zu kompilieren, was vor allem für Entwicklungszwecke gedacht ist, oder für den Fall, dass die zu verwendende Hardware zum Entwicklungszeitpunkt noch nicht bekannt ist. Allerdings führt diese Variante zu längeren Programmlaufzeiten /ADV 24b/.

Durch die Hardwarenähe von ROCm muss der Programmierer Sorge für eine geeignete Speicherverwaltung, die Partitionierung der Aufgabe, die Verteilung der Teilaufgaben und die Synchronisierung tragen /ADV 24b/. ROCm ist mit den meisten AMD Radeon Grafikkarten kompatibel, sowie mit den „Instinct Accelerators“, die extra für HPC Anwendungen konzipiert sind. Die Kompatibilität beschränkt sich aber nicht nur auf AMD GPUs. Die ROCm Plattform ermöglicht durch ihren Aufbau auch die Verwendung von HIP zur Programmierung von NVIDIA GPUs. Neben der Schnittstelle von HIP zum ROCm-Backend, kann intern auch der NVIDIA-Kompilierer „nvcc“ verwendet werden, um HIP Code auf Grafikkarten von NVIDIA ausführbar zu machen. Dies beeinflusst die Geschwindigkeit nicht oder nur in geringem Maß. ROCm ist für Linux und zu großen Teilen auch für Windows verfügbar /ADV 24a/.

4.6 Kokkos

Das Kokkos Ökosystem ist ein Projekt der Linux Foundation, einer nicht-profitorientierten Organisation mit dem Ziel, technologieoffene Projekte zu unterstützen. Kokkos wurde im Jahr 2017 vom Sandia National Laboratory ins Leben gerufen, um ein herstellerunabhängiges, breit anwendbares Programmiersystem für wissenschaftliche C++ Anwendungen auf modernsten HPC Systemen zu entwickeln /SNY 21/. Kokkos basiert auf Bibliotheken und bietet eine breite Anwendbarkeit für verschiedene Hardwarekomponenten und -architekturen /KOK 19/.

Durch die Kokkos C++ Bibliothek werden Funktionalitäten bereitgestellt, die die Parallelisierung von einzelnen Programmteilen erlauben. Es existiert eine breite Kompatibilität mit gängigen C++14 konformen Kompilierern. Eine Liste der Kompilierer ist unter „4. Compiling - Kokkos documentation“ /NAT 24/ zu finden. Es wird empfohlen, die Kompilierung eines Programms über ein CMake Projekt zu steuern, da eine Reihe von hardware- und architekturspezifischen Optimierungen nötig werden können. Dadurch kann ein Programm mit einer Implementierung für sehr viele unterschiedliche Rechnerarchitekturen und Hardware verwendet werden /KOK 19/.

Kokkos bietet Kompatibilität mit den großen Grafikkarten-Herstellern NVIDIA, AMD und Intel /HER 23/. Dabei werden von Kokkos u. a. CUDA, HIP, SYCL aber auch der OpenMP Standard zur Parallelisierung auf CPU und Hardwarebeschleunigern verwendet /KOK 19/.

Da es sich bei Kokkos um eine C++ Bibliothek handelt, die einen C++14 konformen Kompilierer benötigt, kann es für die Parallelisierung von ARTM nötig werden, den Quellcode zu modernisieren. Zum jetzigen Zeitpunkt konnte die GRS nicht abschließend sicherstellen, ob C Quellcode, und damit ARTM, mit der Kokkos Bibliothek vollständig kompatibel ist.

4.7 Zusammenfassung

Im Arbeitspaket 2 wurde recherchiert, welche Standards, Bibliotheken, Kompilierer und Grafikkarten für die Parallelisierung von ARTM zur Verfügung stehen. Dabei beschränkt sich dieser Bericht auf die Parallelisierungsmöglichkeiten, die nach Ansicht der GRS am ehesten das Potential haben, für die Parallelisierung von ARTM geeignet zu sein. Die in diesem Kapitel vorgestellten Standards und Bibliotheken, die die Sichtweise des Programmierers auf Anwendung und die Hardware definieren, werden als parallele Programmiermodelle bezeichnet. Zu den Programmiermodellen, die in diesem Kapitel Erwägung finden, zählen OpenMP, MPI, OpenCL, OpenACC, CUDA, ROCm und Kokkos. Die grundlegenden Eigenschaften dieser Programmiermodelle sind in Tab. 4.1 zusammengefasst. Die meisten der parallelen Programmiermodelle ermöglichen, zumindest bis zu einem gewissen Grad, die Parallelisierung sowohl auf CPU-Basis, also durch die parallele Verwendung von Prozessorkernen eines Mehr-Kern-Prozessors, als auch auf GPU-Basis. Bei GPU-Parallelisierung sind die Anforderungen an die Programmierer, aber auch an die Anwender deutlich erhöht. Die GPU-Parallelisierung bedingt meist die Anwendung von zur individuellen Hardware passenden Treibern. Zumindest aber müssen für die individuelle Hardware geeignete Kompilierer zur Kompilierung des Quellcodes verwendet werden. Aufgrund der Programmstruktur von ARTM und der einfacheren technischen Anforderungen konzentriert sich die GRS daher auf die Parallelisierung auf CPU-Basis.

Tab. 4.1 Übersicht der Eigenschaften der parallelen Programmiermodelle

	OpenMP	MPI	OpenCL	OpenACC	CUDA	ROCm	Kokkos
Typ	Offener Standard	Offener Standard	Offener Standard	Offener Standard	Proprietär	Proprietär	Bibliothek in C++
Parallelisierungsebene	CPU, seit v4.5 auch GPU und FPGA	Inter-CPU, CPU	CPU, GPU und andere Beschleunigerbauteile	GPU und andere Beschleunigerbauteile	GPU	CPU, GPU	CPU, GPU
Unterstützte GPUs	NVIDIA, AMD, Intel; (kompilererabhängig)	-	NVIDIA, AMD, Intel; (hardwarespezifische Treiber nötig)	NVIDIA, AMD, Intel nur indirekt; (kompilerer-abhängig)	NVIDIA (mit proprietärem Kompilerer)	Viele AMD Radeon GPUs, AMD Instinct GPUs, NVIDIA (mit proprietärem Kompilerer	NVIDIA, AMD, Intel
Befehlssteuerung über …	Pragmas	Funktionen	Funktionen, Programmiersprachenerweiterung	Pragmas	Programmiersprachenerweiterung	Programmiersprachenerweiterung	Klassen, Funktionen

5 Arbeitspaket 3: Identifikation möglicher Stellen innerhalb des Quellcodes von ARTM für eine Parallelisierung

Um ARTM sinnvoll parallelisieren zu können, müssen geeignete Stellen im Quellcode identifiziert werden. Ob eine Stelle geeignet ist, hängt dabei von mehreren Faktoren ab. Zum einen muss das Problem, dass an dieser bestimmten Stelle im Quellcode bearbeitet wird, für eine Parallelisierung formal geeignet sein. Zum anderen darf eine Parallelisierung an dieser Stelle nicht die kausalen Abläufe verändern, die für ARTM nötig sind. Außerdem muss eine Parallelisierung zu einem echten Mehrwert, also zu einem effizienteren Programm führen. Eine Parallelisierung ist nur dann sinnvoll, wenn sich daraus ein signifikanter Laufzeitvorteil ergibt.

Zur Identifizierung von Stellen im Quellcode von ARTM, die für eine Parallelisierung von Interesse sind, wurden zwei Methoden verwendet, auf die im Folgenden näher eingegangen wird. Dabei handelt es sich um die Messung der Laufzeiten von Funktionen mit der Hilfe von Profilern und um die Analyse des Programmablaufs von ARTM und TALdia. In diesem Kapitel werden die Ergebnisse dieser Analysen vorgestellt und bewertet.

5.1 Laufzeitmessung von Funktionen im Transportmodell von ARTM

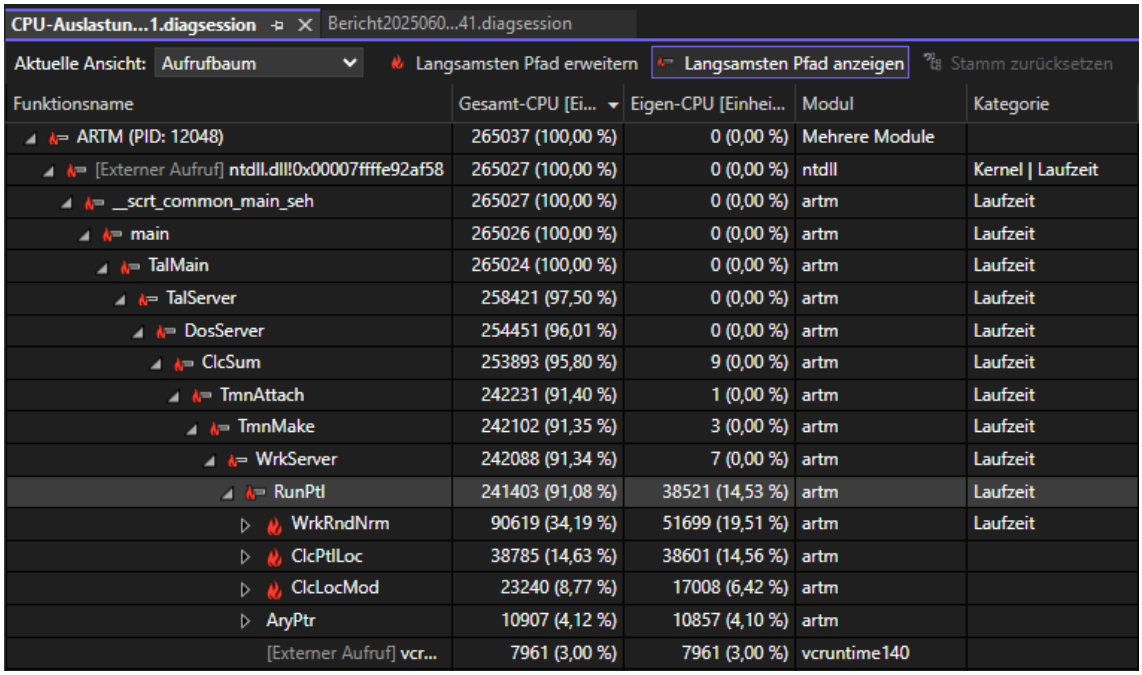
Die Messung von Programm- und Funktionslaufzeiten von ARTM sowie die CPU-Auslastung während der Laufzeit, wurden unter Windows und Linux durchgeführt. Unter Windows wurde der Visual Studio Leistungs-Profiler verwendet /MIC 25/. Unter Linux fand der Profiler des GCC Verwendung /WOL 25/. Dafür wurde ein ARTM-Projekt entworfen, das als Benchmark-Projekt diente. Dieses Projekt ist gerade so komplex gestaltet, dass es als „repräsentatives“ ARTM-Projekt angesehen werden kann, ohne dabei die Programmausführung zu zeitaufwendig werden zu lassen. Die Parametereinstellungen für das Benchmark-Projekt sind in Tab. 5.1 aufgelistet.

Tab. 5.1 Parameter für ARTM Benchmark-Projekt

Merkmal	Wert
Qualitätsstufe der Freisetzungsrate von Partikeln	Standard (0)
Gitterzellenanzahl des Simulationsgebiets	101 x 101 x 19
Maschenweite	50 m
Vertikale Maschenweite	Standardeinstellung
Verdrängungshöhe	0,5 m
Anzahl der Quellen	1
Quellentyp	Punktquelle
Emissionshöhe	50 m
Physikalisch-chemische Eigenschaften	Gas, Aerosol (Klasse 3)
Anemometerhöhe	10 m
Zeitreihe	1 Monat (September)
Dauer eines Zeitschritts	3600 s (1 Std.)

Um die Analyse auf das Partikelmodell von ARTM zu beschränken, wurde bei den ersten Tests auf strukturiertes Gelände oder Gebäude verzichtet. Auf einem Computer mit einem Intel® Core™ i7-1365U der 13. Generation mit 1,80 GHz Taktfrequenz betrug die Laufzeit für den sequenziellen Quellcode von ARTM 4 Minuten und 27 Sekunden.

Der Aufrufbaum der Leistungsanalyse des Visual Studio Leistungs-Profilers von ARTM ist in Abb. 5.1 für den langsamsten Pfad zu sehen. Die markierte Zeile zeigt, dass ca. 91 % der gesamten CPU-Zeit von der Funktion `RunPtl()` verbraucht wurde. Dabei wurden allerdings nur ca. 14 % der gesamten CPU-Zeit von der Funktion selbst verbraucht, während die restlichen 77 % von Funktionen verbraucht wurden, die von `RunPtl()` aufgerufen wurden. Diese Aufteilung spiegelt sich auch im Flammendiagramm in Abb. 5.2 wider. Hier wird der Zeitaufwand für jede Funktion grafisch als Balken dargestellt. Die Balkenlänge entspricht der Laufzeit der Funktion. In den untersten Zeilen ist der Zeitaufwand für die Funktion `RunPtl()` zu sehen. In der darunter liegenden Zeile sind die Zeiten für die Funktionen zu sehen, die von `RunPtl()` aufgerufen werden. Von diesen Funktionen weisen die drei Funktionen `WrkRndNrm()`, `ClcPtlLoc()` und `ClcLocMod()` die größten Zeitverbräuche auf.



Funktionsname	Gesamt-CPU [Ei...	Eigen-CPU [Einhei...	Modul	Kategorie
ARTM (PID: 12048)	265037 (100,00 %)	0 (0,00 %)	Mehrere Module	
[Externer Aufruf] ntdll.dll!0x00007ffffe92af58	265027 (100,00 %)	0 (0,00 %)	ntdll	Kernel | Laufzeit
__scrt_common_main_seh	265027 (100,00 %)	0 (0,00 %)	artm	Laufzeit
main	265026 (100,00 %)	0 (0,00 %)	artm	Laufzeit
TalMain	265024 (100,00 %)	0 (0,00 %)	artm	Laufzeit
TalServer	258421 (97,50 %)	0 (0,00 %)	artm	Laufzeit
DosServer	254451 (96,01 %)	0 (0,00 %)	artm	Laufzeit
ClcSum	253893 (95,80 %)	9 (0,00 %)	artm	Laufzeit
TmnAttach	242231 (91,40 %)	1 (0,00 %)	artm	Laufzeit
TmnMake	242102 (91,35 %)	3 (0,00 %)	artm	Laufzeit
WrkServer	242088 (91,34 %)	7 (0,00 %)	artm	Laufzeit
RunPtl	241403 (91,08 %)	38521 (14,53 %)	artm	Laufzeit
WrkRndNrm	90619 (34,19 %)	51699 (19,51 %)	artm	Laufzeit
ClcPtlLoc	38785 (14,63 %)	38601 (14,56 %)	artm	
ClcLocMod	23240 (8,77 %)	17008 (6,42 %)	artm	
AryPtr	10907 (4,12 %)	10857 (4,10 %)	artm	
[Externer Aufruf] vcr...	7961 (3,00 %)	7961 (3,00 %)	vcruntime140	

Abb. 5.1 Aufrufbaum des langsamsten Pfades der Leistungsanalyse vom Transportmodell von ARTM mit dem Visual Studio Leistungs-Profiler

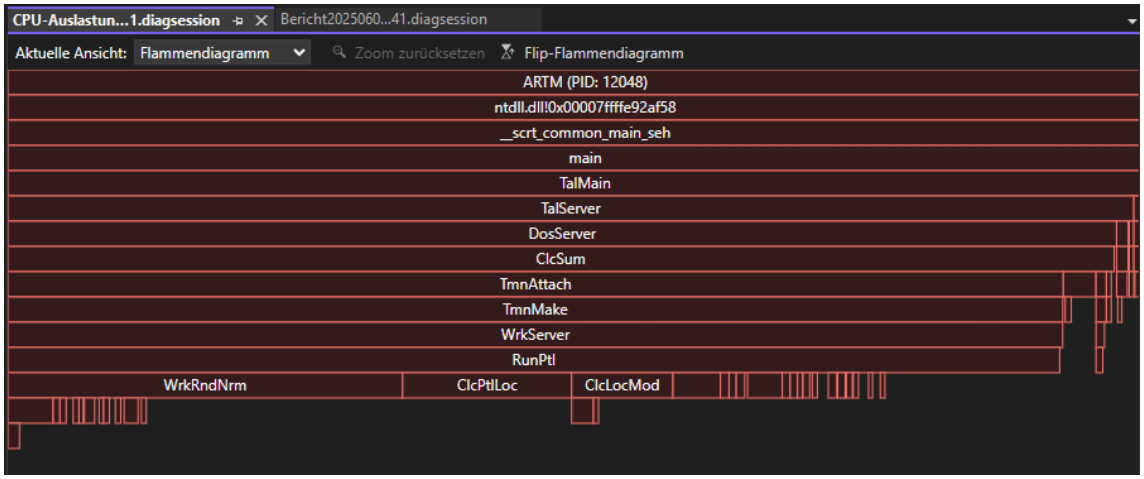


Abb. 5.2 Flammendiagramm des langsamsten Pfades der Leistungsanalyse vom Transportmodell von ARTM mit dem Visual Studio Leistungs-Profiler

Die Funktion `RunPtl()` ist für die Propagation der numerischen Teilchen, die in ARTM modelliert werden, verantwortlich und wird für jedes Teilchen hundertfach durchlaufen, um die Teilchentrajektorien zu berechnen. Das bedeutet allerdings nicht, dass die Funktion `RunPtl()`, mit den von ihr aufgerufenen Funktion, besonders langsam wäre. Es ist viel mehr so, dass die hohe Anzahl der Funktionsdurchläufe für die lange Ausführungszeit verantwortlich ist.

Unter Linux führte die Leistungsanalyse mit GCC zum gleichen Ergebnis, auch wenn für einige Funktionen abweichende Laufzeiten protokolliert wurden. Die Identifizierung der Funktion `RunPtl()` als zeitaufwendigste Funktion bleibt unverändert. Allerdings ist es dem GCC Profiler nicht möglich, Funktionen zu protokollieren, die im Quellcode durch ein Makro anonymisiert wurden. Ein Beispiel für eine solche Anonymisierung ist z. B. in der Datei `TalWrk.c` die Zeile:

```
define WrkRnd()    WrkRndNrm()    /* Normalverteilte Zufalls-
zahlen nehmen. *
```

Die Funktion `WrkRndNrm()` wird vom GCC Profiler als nicht verwendete Funktion klassifiziert, da sie durch das Macro `WrkRnd()` „verdeckt“ wird, wodurch diese im Ergebnisprotokoll nicht aufgeführt wird. Die fehlerhafte Behandlung solcher Funktionen ist vermutlich der Hauptgrund für die Unterschiede der Funktionslaufzeiten zwischen Visual Studio Leistungs-Profiler und GCC Profiler. Die Ergebnisse des GCC Profilers liegen nur in Textform vor. Ein Ausschnitt ist in Abb. 5.3 gegeben.

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 18.27    116.99   116.99 3672613083     0.00     0.00  ClcPtlLoc
 15.77    217.99   101.00    13711     0.01     0.03  RunPtl
 13.76    306.09    88.10 2770131182     0.00     0.00  StdArg
 13.45    392.18    86.09 9141060618     0.00     0.00  AryPtr
 12.34    471.22    79.04 1837035974     0.00     0.00  ClcLocMod

        Call graph (explanation follows)
granularity: each sample hit covers 4 byte(s) for 0.00% of 640.27 seconds
index % time    self  children    called     name
-----------------------------------------------
                              6480             RunPtl <cycle 1> [4]
                             12960             TmnAttach <cycle 1> [32]
                1.92    6.92     751/39701       StdArg [1]
[4]     72.5  101.00  363.38   13711+6480    RunPtl <cycle 1> [4]
              116.99    0.00 3672613083/3672613083     ClcPtlLoc [6]
               79.04   34.60 1837035974/1837035974     ClcLocMod [7]
               88.10    0.00 2770117470/2770131182     StdArg [8]
               44.01    0.00 4672675209/9141060618     AryPtr [9]
                0.32    0.00 10008000/10008000     TalPlm [43]
                0.15    0.11 12036658/22404658     PtlNext [38]
                0.04    0.00   58320/181560      TmnDetach [60]
                0.00    0.01    6480/23749       TmnCreate [95]
                0.01    0.00   12960/19028       DmnGetFloat [117]
                0.01    0.00   12960/88601       TmnInfo [84]
                0.00    0.00   12960/27389       TmnCreator [118]
                0.00    0.00    6480/19440       PtlStart [132]
                0.00    0.00   19440/626166      NmsName [121]
                0.00    0.00    6480/239587      TxtClr [122]
                0.00    0.00   25920/130490      TmString [213]
                0.00    0.00   24024/275961      vMsg [211]
                0.00    0.00   12960/13650       AryAssert [229]
                0.00    0.00    6480/19440       PtlEnd [222]
                             51840             TmnAttach <cycle 1> [32]
                              6480             PtlCount <cycle 1> [37]
                              6480             RunPtl <cycle 1> [4]
-----------------------------------------------
```

```
[…]
-----------------------------------------------
              116.99    0.00 3672613083/3672613083     RunPtl <cycle 1> [4]
[6]     18.3  116.99    0.00 3672613083         ClcPtlLoc [6]
-----------------------------------------------
               79.04   34.60 1837035974/1837035974     RunPtl <cycle 1> [4]
[7]     17.7   79.04   34.60 1837035974         ClcLocMod [7]
               34.60    0.00 3674071948/9141060618     AryPtr [9]
-----------------------------------------------
```

Abb. 5.3 Ausschnitte der Analysedatei des GCC Profilers für ARTM unter Linux

Bei „flat profile“ ist die erste Spalte die Laufzeit der Funktion in Prozent von der Programmlaufzeit, die zweite Spalte die Summe der Laufzeit der Funktion und der darüber gelisteten Funktionen in Sekunden, die dritte Spalte die Laufzeit der Funktion in Sekunden, die vierte Spalte die Anzahl der Aufrufe, die fünfte Spalte die Laufzeit der Funktion pro Aufruf in Millisekunden, die sechste Spalte die durchschnittliche Laufzeit der Funktion und der von dieser aufgerufenen Funktionen pro Aufruf in Millisekunden und in der siebten Spalte der Funktionsname. Im „call graph“ ist in der ersten Spalte ein Index, in der zweiten Spalte die Laufzeit der Funktion und den im Aufrufstapel nachgeordneten Funktionen in Prozent der Programmlaufzeit, in der dritten Spalte die gesamte Laufzeit dieser Funktion, in der vierten Spalte die Laufzeit, die von den unmittelbar nachgeordneten Funktionen verbraucht wurde, in der fünften Spalte die Anzahl der Aufrufe und in der sechsten Spalte der Funktionsname gegeben.

5.2 Analyse des Programmablaufs des Transportmodells von ARTM

Der Programmablauf des Transportmodells von ARTM lässt sich, wie fast jedes Computerprogramm, in die drei Teile Dateneingabe, Datenverarbeitung und Datenausgabe[13] unterteilen. Davon ist die Datenverarbeitung der Teil, der sich durch eine Parallelisierung mit den größten Erfolgsaussichten optimieren lässt.

Nach der erfolgreichen Dateneingabe und der Erstellung des Windfeldes besteht die wesentliche Tätigkeit von ARTM darin, die numerischen Teilchen zu propagieren. ARTM geht dabei schrittweise jeden Zeitschritt, der in der Zeitreihendatei enthalten ist, durch. Bei jedem dieser Zeitschritte, die typischerweise eine Dauer von 3600 s (1 Std.) haben, wird eine Schleife über die zu berechnenden Rechengitter (nur bei geschachtelten Gittern) durchlaufen. Für jedes Gitter werden dann neun Simulationsgruppen erstellt. Jeder dieser Simulationsgruppen wird 1/9 der numerischen Teilchen zugeordnet, die in diesem Zeitschritt zu emittieren sind. Im nächsten Zeitschritt werden die bereits vorhandenen Teilchen weitergeführt. Effektiv ergeben sich somit nach der Abarbeitung der gesamten Zeitreihe also neun statistisch unabhängige Ausbreitungssimulationen, die für die

[13] Das EVA-Prinzip steht für Eingabe, Verarbeitung und Ausgabe von Daten.

statistische Auswertung in ARTM verwendet werden. In einer Simulationsgruppe werden alle numerischen Teilchen der Reihe nach, entsprechend der Zeitschrittdauer, durch das Rechengebiet propagiert. Das nächste Teilchen wird erst propagiert, wenn das vorherige Teilchen den gesamten Zeitschritt durchlaufen hat.

Ausgehend von diesem Programmablauf ergeben sich Einschränkungen für eine Parallelisierung von ARTM. Die einzelnen Zeitschritte oder mehrere aufeinanderfolgende Gruppierungen von Zeitschritten der Zeitreihe lassen sich aus Kausalitätsgründen nicht ohne Informationsverlust parallelisieren. Bei einer Parallelisierung der Zeitschritte würden die numerischen Teilchen eines Zeitschritts bei der parallelen Abarbeitung des darauffolgenden Zeitschritts fehlen, da der Übertrag der numerischen Teilchen vom einen in den anderen Zeitschritt nicht erfolgen kann.

Die parallele Abarbeitung von geschachtelten Rechengittern birgt ebenfalls Kausalitätsprobleme, da die numerischen Teilchen, die ein Gitter verlassen, in das nächstgröbere Gitter transferiert werden und dort weiter propagiert werden. Ebenso ist zu beachten, dass ARTM Simulationen nicht immer mit geschachteltem Gitter verwendet werden und wenn doch, dann bewegt sich deren Anzahl im niedrigen bis mittleren einstelligen Bereich. Eine Laufzeitverkürzung und die Auslastung der gesamten CPU-Leistung würde damit linear von der Anzahl der geschachtelten Rechengitter abhängen. Wenn keine Gitterschachtelung verwendet wird, würde es zu keiner Laufzeitverkürzung kommen.

Bei einer parallelen Abarbeitung von Simulationsgruppen ergeben sich keine Kausalitätsprobleme. Da in ARTM, unabhängig von der genauen Projektkonfiguration, immer neun Simulationsgruppen verwendet werden, würde ein sehr breiter Bereich von Anwendungsfällen von dieser Parallelisierung profitieren können.

Ähnlich sollte es sich auch bei der Parallelisierung auf der Ebene der Teilchenpropagation verhalten. Da die einzelnen Teilchen als voneinander unabhängig modelliert werden, sind Kausalitätsprobleme auszuschließen. Die Anzahl der numerischen Teilchen ist in jedem Fall groß genug, um auch bei sehr einfachen ARTM-Projekten, ausreichend Potential für eine Laufzeitverbesserung zu bieten.

5.3 Laufzeitmessung von Funktionen in TALdia

Für die Messung von Programm- und Funktionslaufzeiten von TALdia unter Windows und Linux sowie die CPU-Auslastung während der Laufzeit, wurde das Benchmark-

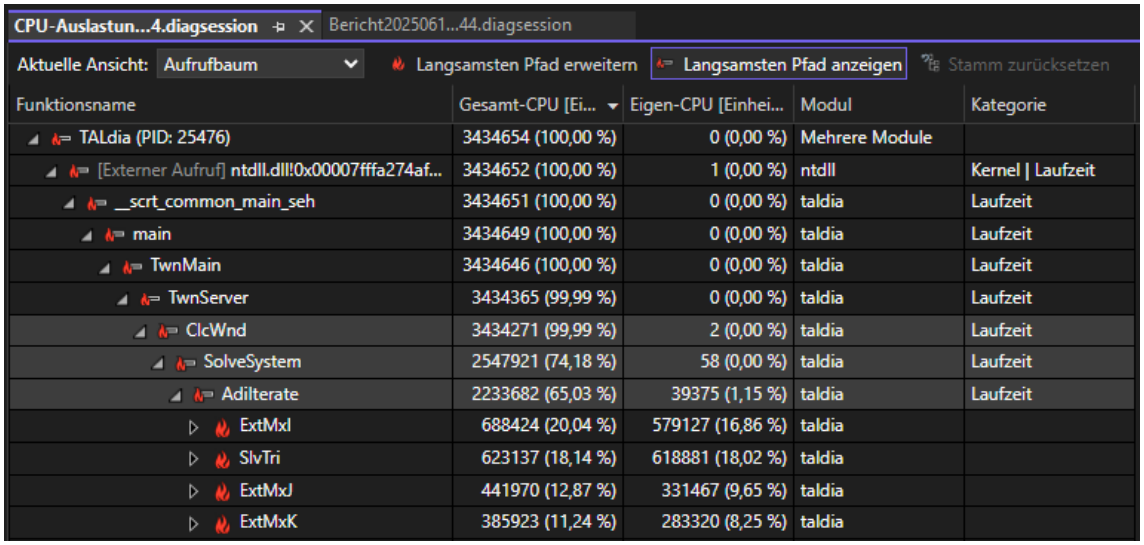
Projekt aus Abschnitt 5.1 um eine Geländedatei und ein Gebäude erweitert. Die grundlegenden Parameter des Benchmark-Projekts sind in Tab. 5.1 aufgelistet. Bei der Geländedatei handelt es sich um `tittling.grid`. Die Parameter für das Gebäude sind in Tab. 5.2 gegeben. Die Verwendung eines Gebäudes bedingt standardmäßig ein feineres vertikales Gitter mit 29 anstatt 19 vertikalen Schichten.

Tab. 5.2 Gebäudeparameter für das verwendete Gebäude

Merkmal	Wert
Rechtswert der Position	6000 m
Hochwert der Position	2175 m
Gebäudelänge	100 m
Gebäudebreite	60 m
Gebäudehöhe	20 m
Drehwinkel	70°

Auf einem Computer mit einem Intel® Core™ i7-1365U der 13. Generation mit 1,80 GHz Taktfrequenz betrug die Laufzeit für den sequenziellen Quellcode von TALdia 58 Minuten und 21 Sekunden.

Der Aufrufbaum des Visual Studio Leistungs-Profilers unter Windows für den langsamsten Pfad der Leistungsanalyse von TALdia ist in Abb. 5.4 zu sehen. Die markierten Zeilen zeigen die Funktionen mit dem meisten CPU-Gesamtverbrauch. Die Funktion `ClcWnd()` ist für nahezu 100 % der verbrauchten CPU-Zeit verantwortlich, wobei die Funktion selbst keinen nennenswerten Arbeitsaufwand hat. Sie ruft aber weitere Funktionen auf, die für die hohe CPU-Zeit verantwortlich sind. Die zeitintensivste Funktion ist `SolveSystem()`, die wiederum `AdiIterate()` aufruft. Auch diese Funktionen weisen nur einen irrelevanten Eigenverbrauch auf, rufen ihrerseits aber weitere Funktionen auf, die über einen nennenswerten Verbrauch von CPU-Zeit verfügen.



Funktionsname	Gesamt-CPU [Ei...	Eigen-CPU [Einhei...	Modul	Kategorie
TALdia (PID: 25476)	3434654 (100,00 %)	0 (0,00 %)	Mehrere Module	
[Externer Aufruf] ntdll.dll!0x00007fffa274af...	3434652 (100,00 %)	1 (0,00 %)	ntdll	Kernel | Laufzeit
__scrt_common_main_seh	3434651 (100,00 %)	0 (0,00 %)	taldia	Laufzeit
main	3434649 (100,00 %)	0 (0,00 %)	taldia	Laufzeit
TwnMain	3434646 (100,00 %)	0 (0,00 %)	taldia	Laufzeit
TwnServer	3434365 (99,99 %)	0 (0,00 %)	taldia	Laufzeit
ClcWnd	3434271 (99,99 %)	2 (0,00 %)	taldia	Laufzeit
SolveSystem	2547921 (74,18 %)	58 (0,00 %)	taldia	Laufzeit
AdiIterate	2233682 (65,03 %)	39375 (1,15 %)	taldia	Laufzeit
ExtMxI	688424 (20,04 %)	579127 (16,86 %)	taldia	
SlvTri	623137 (18,14 %)	618881 (18,02 %)	taldia	
ExtMxJ	441970 (12,87 %)	331467 (9,65 %)	taldia	
ExtMxK	385923 (11,24 %)	283320 (8,25 %)	taldia	

Abb. 5.4 Aufrufbaum des langsamsten Pfades der Leistungsanalyse von TALdia mit dem Visual Studio Leistungs-Profiler

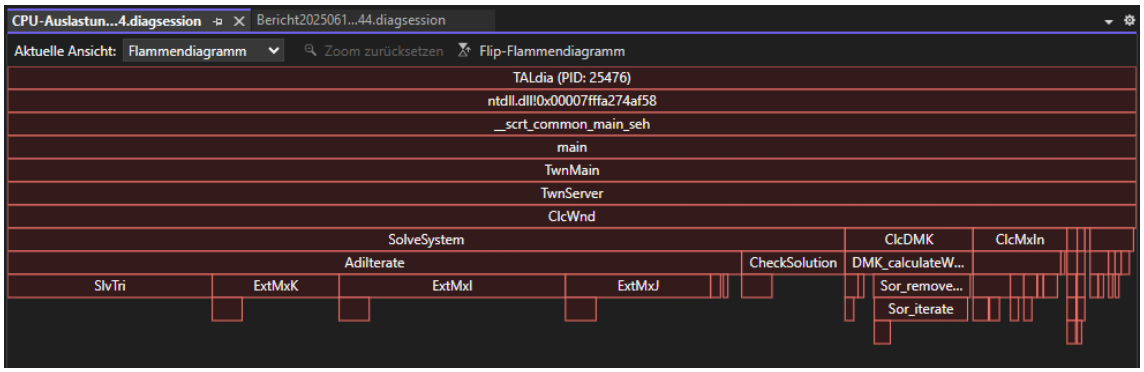


Abb. 5.5 Flammendiagramm des Langsamsten Pfades der Leistungsanalyse von TALdia mit dem Visual Studio Leistungs-Profiler

Ein genaueres Bild der aufgerufenen Funktionen lässt sich aus dem Flammendiagramm in Abb. 5.5 gewinnen. Die Funktion `AdiIterate()` ruft die vier Funktionen `SlvTri()`, `ExtMxK()`, `ExtMxI()` und `ExtMxJ()` auf, die für die hohen CPU-Zeitverbräuche maßgeblich verantwortlich sind. Alle Funktionen, die von `SolveSystem()` aufgerufen werden sind für die iterative Lösung der Gleichungssysteme zuständig, die für die Berechnung der Windfelder benötigt werden. Diese müssen sequenziell ablaufen, so dass sich hier kein Potential für eine Parallelisierung erkennen lässt.

Der GCC Profiler identifiziert unter Linux ebenfalls die Funktion `ClcWnd()` als diejenige mit dem höchsten CPU-Zeitverbrauch, wobei die von dieser Funktion aufgerufenen Funktionen die eigentlichen Zeitkonsumenten sind. Diese Erkenntnisse stimmen mit denen des Visual Studio Leistungs-Profilers unter Windows überein. Allerdings können

einige der Funktionen, die die Profiler als Funktionen mit großen CPU-Zeitverbräuchen identifizieren, durch kompiliererseitige Optimierung als vernachlässigbar eingestuft werden. Das betrifft die Funktionen `ExtMxK()`, `ExtMxI()` und `ExtMxJ()`. Dennoch bleibt die Funktion `ClcWnd()` mit den von ihr aufgerufenen Funktionen die, die die meiste CPU-Zeit verbraucht. Ein Auszug des Ergebnisprotokolls ohne Kompiliereroptimierung ist in Abb. 5.6 dargestellt.

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 30.15      0.81     0.81 164191212     0.00     0.00  AryPtrX
 10.86      1.10     0.29   493240     0.00     0.00  ExtMxI
  9.74      1.36     0.26  1505680     0.00     0.00  SlvTri
  8.99      1.60     0.24     1346     0.00     0.00  CheckSolution
  8.99      1.84     0.24  3772800     0.00     0.00  AddLform
  7.87      2.05     0.21   519200     0.00     0.00  ExtMxK
  7.87      2.26     0.21   493240     0.00     0.00  ExtMxJ
  3.00      2.34     0.08                             brk
  2.25      2.40     0.06  1681986     0.00     0.00  AryPtr
  2.25      2.46     0.06   537600     0.00     0.00  ClcVelS
  1.12      2.49     0.03     1298     0.00     0.00  AdiIterate
  1.12      2.52     0.03                             __profile_frequency

        Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 0.37% of 2.67 seconds
index % time    self  children    called     name
-----------------------------------------------
                0.00    2.44      12/12          TwnServer [4]
[5]     91.4    0.00    2.44      12         ClcWnd [5]
                0.00    1.64      24/24          SolveSystem [6]
                0.00    0.47      24/24          ClcMxIn [9]
                0.02    0.24      36/36          ClcUvw [17]
                0.00    0.03      24/24          UvwInit [25]
                0.00    0.02      24/24          writeDmna [28]
                0.00    0.01      12/12          InsertLayer [30]
                0.00    0.01      72/507         TmnAttach <cycle 1> [60]
                0.00    0.00      12/36          DefGrid [29]
                0.00    0.00      12/12          ReadInit [44]
                0.00    0.00      12/12          ClcUref [45]
                0.00    0.00      12/12          ClcCharHL [50]
                0.00    0.00      48/802         TmnDetach [91]
                0.00    0.00      48/108         TmnDelete [132]
                0.00    0.00      36/2136        TxtForm [83]
                0.00    0.00      36/7701        TxtCat [76]
                0.00    0.00      24/24          ClrArrays [164]
                0.00    0.00      24/527         DmnRplValues [97]
                0.00    0.00      24/2446        NmsName [81]
                0.00    0.00      24/1957        TxtClr [84]
                0.00    0.00      12/2563        vMsg [80]
                0.00    0.00      12/12          TmnRename [195]
                0.00    0.00      12/331         TxtCpy [102]
-----------------------------------------------
                0.00    1.64      24/24          ClcWnd [5]
[6]     61.4    0.00    1.64      24         SolveSystem [6]
                0.03    1.26    1298/1298        AdiIterate [7]
                0.24    0.10    1346/1346        CheckSolution [13]
                0.00    0.01      72/507         TmnAttach <cycle 1> [60]
                0.00    0.00    1442/2563        vMsg [80]
                0.00    0.00      72/802         TmnDetach [91]
-----------------------------------------------
                0.03    1.26    1298/1298        SolveSystem [6]
```

```
[7]     48.3    0.03    1.26    1298         AdiIterate [7]
                0.29    0.10  493240/493240      ExtMxI [11]
                0.21    0.10  519200/519200      ExtMxK [14]
                0.21    0.10  493240/493240      ExtMxJ [15]
                0.26    0.00 1505680/1505680     SlvTri [16]
-----------------------------------------------
```

Abb. 5.6 Ausschnitte der Analysedatei des GCC Profilers für TALdia unter Linux

Für die Beschreibung der Spalten siehe Abb. 5.3.

5.4 Analyse des Programmablaufs von TALdia

Im Vergleich zum Transportmodell von ARTM ist der Programmablauf in TALdia weniger stark geschachtelt. Nach dem Einlesen aller relevanten Eingabeparameter wird von TALdia berechnet, wie viele Windfelder für die angegebenen Eingabeparameter berechnet werden sollen. Falls lediglich mit Gelände, aber ohne Gebäude, gerechnet wird, werden von TALdia zwei Windfelder für jede in der Zeitreihe vorkommende Ausbreitungsklasse berechnet. Diese werden automatisch gewählt und stehen senkrecht zueinander. Falls die Eingabeparameter Gebäude vorsehen, werden für jede Ausbreitungsklasse, die in der Zeitreihendatei vorkommt, 36 Windfelder berechnet. Diese weisen Richtungen in 10°-Schritten von 10 – 360° auf. Somit ergeben sich bei TALdia mindestens zwei Windfelder (wenn die Zeitreihe nur einen Zeitschritt beinhaltet) und höchstens 6 x 36 = 216 Windfelder. Für jedes dieser Windfelder wird eine Schleife über die geschachtelten Simulationsgittern durchlaufen. Es wird also für jede vorher genannte Windrichtung, jede Ausbreitungsklasse und jedes geschachtelte Gitter ein Windfeld berechnet, so dass sich die Anzahl der Windfelder nochmals entsprechend der Anzahl der geschachtelten Gitter erhöht. Anschließend werden alle Windfelder von geschachtelten Gittern derselben Windsituation als dmna- bzw. dmnb-Datei auf die Festplatte geschrieben.

Simulationen mit wenigen Simulationszeitschritten, konstanter Turbulenz und dem Verzicht auf Gebäude weisen in der Regel eine kurze Rechenzeit auf. Abgesehen von diesen Simulationen ergibt sich sehr schnell eine Anzahl von Windfeldern, die für die parallele Abarbeitung auf CPU-Basis geeignet sind. Somit ist ein Ansatzpunkt zur Parallelisierung von TALdia die Parallelisierung der Schleife, die die Windfelder berechnet.

Der verbleibende Ansatzpunkt für eine Parallelisierung ist die Schleife, die die Windfelder der geschachtelten Gitternetze berechnet. Es werden allerdings nicht bei allen Simulationen geschachtelte Gitternetze verwendet, und deren Anzahl beschränkt sich meist auf einen niedrigen bis mittleren einstelligen Bereich. Eine Laufzeitverkürzung und

die Auslastung der gesamten CPU-Leistung würde damit linear von der Anzahl der geschachtelten Rechengitter abhängen. Wenn keine Gitterschachtelung verwendet wird, würde es zu keiner Laufzeitverkürzung kommen.

5.5 Konzept zur Parallelisierung von ARTM

Ausgehend von den Rechercheergebnissen aus den Kapiteln 3 und 4 sowie den Laufzeit- und Programmablaufanalysen aus den Abschnitten 5.1 bis 5.4 wurde ein Konzept für die Parallelisierung von ARTM erarbeitet, das im Folgenden vorgestellt wird. Dabei wurde besonderes Augenmerk darauf gelegt, die vorliegenden Teilaufgaben des Programms effizient auf die gegebene Hardware zu verteilen.

Bei der Parallelisierung auf GPU-Basis ergeben sich einige erschwerende Besonderheiten. Die GPU-Parallelisierung ist in nahezu allen Fällen in einem hohen Maße von der verwendeten Hardware abhängig. Darüber hinaus ist eine Parallelisierung auf GPU-Basis nur sinnvoll, wenn die Struktur des zu lösenden Problems für eine GPU-Parallelisierung geeignet ist. Da in ARTM kaum datenparallele Problemstellungen vorliegen, die für eine GPU-Parallelisierung besonders geeignet wären, wird die GPU-Parallelisierung zu Gunsten einer CPU-Parallelisierung von ARTM aufgegeben.

Für die CPU-Parallelisierung wurde OpenMP als paralleles Programmiermodel gewählt. OpenMP ist ein offener, weit verbreiteter Standard, der seit Jahren kontinuierlich weiterentwickelt wird und von den meisten gängigen Kompilierern in verschiedenen Versionen unterstützt wird /OPE 19/. Die breite Liste von internationalen Unterstützern dieses Standards lässt darauf schließen, dass OpenMP noch viele Jahre weiterbetrieben wird, so dass auch eine zukünftige Weiterentwicklung dieses Programmiermodells erwartet werden kann. Darüber hinaus verfügt OpenMP auch über die Möglichkeit einer GPU-Parallelisierung. Somit bleiben Optimierungen der Parallelisierung unter Einbezug von GPUs weiterhin möglich.

ARTM lässt sich in zwei Teilmodelle unterteilen, das Windfeldmodell TALdia, das für die Berechnung der diagnostischen Windfelder verantwortlich ist und das Transportmodell, das die Propagation der numerischen Teilchen gemäß den Windfeldern und der Partikeleigenschaften realisiert. In beiden Teilen konnte zumindest je eine Stelle identifiziert werden, die sich potenziell für eine Parallelisierung eignet.

5.5.1 Parallelisierungsansatz für das Transportmodell von ARTM

Für das Transportmodell von ARTM konnten zwei mögliche Ansatzpunkte identifiziert werden. Der erste Ansatzpunkt zur Parallelisierung bezieht sich auf die neun Simulationsgruppen. Diese werden in der Funktion `ClcSum()` der Datei `TalDos.c` durchlaufen. Hier existiert eine `for`-Schleife, die die Propagation der numerischen Teilchen für die neun Simulationsgruppen steuert. Diese `for`-Schleife kann durch OpenMP Präprozessor Direktiven parallelisiert werden. Dazu muss sichergestellt werden, dass jeder parallele Zugriff auf globale Variablen fehlerfrei abläuft. Unter Umständen müssen dafür Kopien dieser Variablen für jede parallel ablaufende Simulationsgruppe erstellt werden.

Für die maximale theoretische Beschleunigung des Transportmodells von ARTM ergibt sich für diese Parallelisierungsstrategie nach dem Amdahlschen Gesetz ein Faktor von

$$\eta_{B,\text{Gruppen}} = \frac{t_s + t_p}{t_s + \frac{t_p}{n_p}} = \frac{4{,}20 + 95{,}80}{4{,}20 + \frac{95{,}80}{9}} \approx 6{,}74. \tag{5.1}$$

Dabei ist n_p durch die Anzahl der neun Simulationsgruppen festgelegt. Aufgrund des nicht berücksichtigten Synchronisierungsaufwands ist mit einer geringeren Beschleunigung zu rechnen.

Der zweite Ansatzpunkt bezieht sich auf die Parallelisierung der Teilchenpropagation. Hierbei wird in der Funktion `RunPtl()` der Datei `TalWrk.c` die `for`-Schleife mit OpenMP Präprozessor Direktiven parallelisiert, die die Propagation der einzelnen numerischen Teilchen steuert, so dass mehrere Teilchen parallel propagiert werden können. Auch hierbei ist darauf zu achten, dass der Zugriff auf globale Variablen fehlerfrei organisiert wird. Dazu müssen für einige Variablen Kopien für die parallel ablaufenden Prozesse erstellt werden. Bei der Umsetzung dieses Parallelisierungsansatzes ist nach dem Amdahlschen Gesetz eine maximale theoretische Beschleunigung von

$$\eta_{B,\text{Partikel}} = \frac{t_s + t_p}{t_s + \frac{t_p}{n_p}} = \frac{8{,}92 + 91{,}08}{8{,}92 + \frac{91{,}08}{12}} \approx 6{,}06 \tag{5.2}$$

erreichbar. Auch hier ist aufgrund von nicht berücksichtigtem Synchronisierungsaufwand mit geringeren Beschleunigungen zu rechnen. Für die Berechnung wurde die Parallelisierung auf 12 logischen Prozessorkernen angenommen.

In beiden Fällen existiert ein Kausalitätsproblem mit der Ziehung der Zufallszahlen, die für die Propagation der numerischen Teilchen nötig sind. In der Funktion `RunPtl()` werden an mehreren Stellen Zufallszahlen für die Positionierung der Teilchen in der Emissionsquelle und die turbulenten Windgeschwindigkeiten gezogen. Bei diesen Zufallszahlen handelt es sich um Pseudo-Zufallszahlen, die sich bei einem gegebenen Initialwert (engl. „seed“) deterministisch berechnen. Die Reihenfolge dieser Zufallszahlen folgt einem kausalen Zusammenhang, der bei einer parallelen Behandlung des Transportmodells von ARTM unterbrochen wird. Bei einer Parallelisierung des Transportmodells wird die Reihenfolge der propagierenden Teilchen zufällig geändert, je nachdem, welche parallelen Prozesse zuerst ablaufen. Das hat zur Folge, dass die Reproduktion von Konzentrationsfeldern einer ARTM Simulation nicht mehr möglich ist.

Dieses Problem könnte umgangen werden, wenn für beide Ansätze individuelle Strategien zur Ziehung der Zufallszahlen implementiert werden. Für die Parallelisierung der Simulationsgruppen müsste jede Gruppe einen eigenen Initialwert für die Zufallszahlen dieser Simulationsgruppe bekommen, so dass jede Gruppe ihre eigenen, privaten Zufallszahlen zieht. Das würde die Reproduzierbarkeit von Simulationsergebnissen von ARTM wieder herstellen. Allerdings würde sich dadurch keine Übereinstimmung zwischen den Simulationsergebnissen mit der sequenziellen und parallelen ARTM Version ergeben.

Für die Parallelisierung der Teilchenpropagation ist diese Lösung ebenfalls anwendbar. Dafür muss gegebenenfalls eine durchlaufende Nummerierung der numerischen Teilchen eingeführt werden. Allerdings kann es aufgrund der Anzahl der Simulationsteilchen zu einem großen Mehraufwand für die Bereitstellung der Zufallszahlen kommen, da für jedes numerische Teilchen ein eigener Zufallszahlengenerator initialisiert werden muss.

5.5.2 Parallelisierungsansatz für das Windfeldmodell TALdia

In TALdia befindet sich in der Datei `TalWnd.c` in der Funktion `TwnMain()` eine `while`-Schleife, die die Schleife über alle zu berechnenden Windfelder darstellt. Ziel ist es, diese `while`-Schleife in eine `for`-Schleife umzuschreiben, die dann mit den entsprechenden OpenMP Präprozessor Direktiven parallelisiert werden kann. Dazu ist es

erforderlich, die Berechnung einiger Variablen so zu ändern, dass diese nicht mehr vom Ergebnis einer vorherigen Windfeldberechnung abhängen. Zudem muss sichergestellt werden, dass jeder parallele Zugriff auf globale Variablen fehlerfrei abläuft. Unter Umständen müssen dafür Kopien dieser Variablen für jede parallel ablaufende Windfeldberechnung erstellt werden.

Bei einer erfolgreichen Parallelisierung würde sich ein maximal theoretischer Beschleunigungsfaktor gemäß des Amdahlschen Gesetzes von

$$\eta_{B,\mathrm{TALdia}} = \frac{t_s + t_p}{t_s + \frac{t_p}{n_p}} = \frac{0{,}01 + 99{,}99}{0{,}01 + \frac{99{,}99}{12}} \approx 11{,}99 \tag{5.3}$$

ergeben, wobei t_s der angenommene serielle Anteil des Programms, t_p der angenommene parallele Anteil des Programms und n_p die Anzahl der Prozessorkerne ist. Die Werte für t_s und t_p stammen aus den Ergebnissen der Laufzeitprofilierung. Für die Berechnung in Gl. (5.3) wurde die Parallelisierung auf 12 logischen Prozessorkernen angenommen. Da nahezu 100 % des Quellcodes von TALdia parallelisierbar ist, kann jedoch in erster Näherung davon ausgegangen werden, dass die Beschleunigung mit einer beliebigen Anzahl der Kerne linear skaliert. Die Skalierbarkeit ist nach oben hin durch die Anzahl der zu berechnenden Windfelder limitiert. Zu beachten ist, dass die Laufzeiten für die Synchronisierungsaufwände nicht berücksichtigt sind. Ebenso sind Eingabe-/Ausgabeoperationen oder zwingend sequenzielle Programmabläufe innerhalb des angenommenen parallelen Anteils des Programms nicht berücksichtigt, so dass die reale Beschleunigung durch die Parallelisierung geringer ausfallen wird.

5.6 Zusammenfassung

Um in ARTM mögliche Stellen im Quellcode zu identifizieren, die für eine Parallelisierung geeignet sind, wurden Leistungsprofilierungen und Programmablaufanalysen durchgeführt. Für die Leistungsprofilierung wurde unter Windows der Leistungsprofiler von Microsoft Visual Studio Professional verwendet, während unter Linux der Leistungsprofiler des GCC Kompilierers Anwendung fand. Die Programmablaufanalyse wurde dadurch erstellt, dass der Quellcode gelesen und schrittweise ausgeführt wurde.

Auf der Grundlage dieser Analysen wird für das Windfeldmodell TALdia die Parallelisierung der Berechnung der einzelnen Windfelder favorisiert. Hierfür ist im Wesentlichen die Parallelisierung einer `while`-Schleife nötig. Da sich diese Schleife sehr weit oben im Aufrufstapel befindet, werden wahrscheinlich zahlreiche Anpassungen von Funktionen nötig werden, die sich im Aufrufstapel weiter unten befinden. Da für die Berechnung des Windfeldes nahezu die gesamte Programmlaufzeit aufgewendet wird, kann theoretisch mit einer Beschleunigung gerechnet werden, die fast der Anzahl der verwendeten Prozessorkerne entspricht. Hierbei ist der Overhead allerdings nicht berücksichtigt, so dass eine geringere Beschleunigung zu erwarten ist.

Für das Transportmodell ARTM konnten zwei verschiedene Ansätze identifiziert werden. Einerseits besteht die Möglichkeit, durch die Parallelisierung einer `for`-Schleife, die Abarbeitung der neun Simulationsgruppen zu parallelisieren. Andererseits wäre es möglich, die Propagation der numerischen Partikel zu parallelisieren. Dafür müsste im Wesentlichen ebenfalls eine `for`-Schleife für die parallele Ausführung modifiziert werden. Beide Ansätze scheinen eine ähnliche theoretische Laufzeitverbesserung zu versprechen, wobei die Skalierbarkeit der Beschleunigung für den Ansatz der Parallelisierung der Simulationsgruppen begrenzt ist. Die Parallelisierung der Teilchenpropagation weist praktisch keine Begrenzung der Skalierbarkeit auf. Dadurch, dass im Vergleich zur Parallelisierung der neun Simulationsgruppen, die Parallelisierung der Teilchenpropagation weiter unten im Aufrufstapel erfolgt, ist hier mit weniger Implementierungs- und Änderungsaufwand zu rechnen. Somit wird für dieses Eigenforschungsvorhaben die Parallelisierung der Teilchenpropagation favorisiert.

6 Arbeitspaket 4: Analyse von Implementierungsmethoden und daraus folgender Anpassungsbedarf in ARTM

In den folgenden Abschnitten werden mögliche Implementierungsmethoden und erste Ergebnisse einer parallelisierten Version von ARTM/TALdia präsentiert, welche auf den im Arbeitspaket 3 identifizierten Stellen innerhalb des Quellcodes aus dem Kapitel 5 aufbauen.

6.1 Umsetzung der Parallelisierung in ARTM auf der Ebene der Teilchenpropagation

Die Parallelisierung in ARTM auf der Ebene der Teilchenpropagation gemäß dem erarbeiteten Konzept setzt sich aus mehreren einzelnen Aspekten zusammen, die in den folgenden Abschnitten einzeln erläutert werden. Im Abschnitt 6.1.1 werden die allgemeinen Voraussetzungen für die Anwendung von OpenMP bereitgestellt. Dazu gehört die Anpassung von Variablen sowie die Bereitstellung und Analyse der parallelen Umgebung. Die Abschnitte 6.1.2 und 6.1.3 präsentieren typische Implementierungsfehler innerhalb des Parallelisierungsprozesses eines sequenziellen Programms, analysiert diese und nennt deren Fehlerbehebung.

6.1.1 Allgemeine Anpassungen und Bereitstellung der parallelen Umgebung

ARTM und auch TALdia arbeiten sehr viel mit **globalen Variablen**. Bei einer Parallelisierung mittels OpenMP muss sichergestellt werden, dass diese globalen Variablen nicht innerhalb der Ebene der Teilchenpropagation geändert werden. Hintergrund ist, dass jeder parallele Thread den Wert für einen anderen Thread zu jeder Zeit ändern kann. Setzt beispielsweise der Thread 1 eine globale Variable `random=1` und will diese zu einem späteren Zeitpunkt innerhalb der Schleifenebene nutzen, kann es passieren, dass in der Zwischenzeit Thread 2 diese globale Variable auf `random=2` setzt. Um diese Problematik zu umgehen, gibt es mehrere Möglichkeiten. Zu Beginn einer parallelen Umgebung können OpenMP Präprozessor Direktiven für jeden Thread eine private Kopie dieser globalen Variable erstellen. Eine weitere Möglichkeit, die z. B. zur Erzeugung der zufälligen Zahlen für den turbulenten Transport der Teilchen verwendet wurde, ist die Anpassung der Variablen zu Arrays. Durch Übergabe der entsprechenden Thread-Nummer in der parallelen Umgebung ändert somit jeder Thread seine eigene globale Variable innerhalb des Arrays.

Ein weiterer wichtiger Faktor war die Untersuchung der Zuteilung des Speichers der einzelnen Strukturen und Variablen für die parallele Umgebung und die damit einhergehende Initialisierung der parallelen Umgebung. Der **Overhead**, d. h. der zusätzliche Aufwand zur Zerlegung, Verwaltung und späteren Zusammenführung der einzelnen parallelen Threads, spielt dabei eine essenzielle Rolle. Dieser wurde am Beispiel des Allozierens der Teilchen und der Struktur `PtlRec`, welches die Information der zu transportierenden Teilchen enthält, untersucht. Dabei wurden drei Varianten betrachtet und mittels Benchmark-Tests verglichen. Variante A basiert auf einer Allokation für jedes Teilchen in der parallelen Umgebung, während Variante B auf einer Allokation außerhalb der Teilchenschleife innerhalb eines parallelen Blocks aufbaut.

Eine dritte Variante C allokiert einmal im Hauptthread (Vorausallokation) einen Speicherblock, statt dynamisch in jedem Thread separat. Jeder Thread erhält dann in der parallelen Umgebung seinen eigenen Block. Letztere Methode ist insbesondere dann sinnvoll, wenn die parallele Umgebung sehr oft erzeugt wird und die Allokation der Strukturen einen signifikanten Anteil am Overhead haben. Es zeigte sich jedoch, dass der Overhead aller drei Methoden im Vergleich zu dem komplexen Transport innerhalb der Partikelschleife vernachlässigbar ist. Die Variationen des Overheads der drei Methoden lagen untereinander in der Größenordnung von etwa 10 % und pro Gruppe und Zeitschritt in der Größenordnung von $(0.2 - 10)\text{x}10^5$ ns. Die komplette Funktion `RunPtl()` pro Gruppe und Zeitschritt betrug hingegen $(0.5 - 10)\text{x}10^7$ ns. Der Overhead ist somit in etwa zwei Größenordnungen geringer als der Transport innerhalb der Partikelschleife. Aus diesem Grund wurde von einer weiteren Analyse des Overheads abgesehen.

Ebenfalls wurde überprüft, ob das Programm tendenziell **memory-Bound oder compute-Bound** ist. Die Partikelschleife wäre memory-Bound, wenn die benötigte Zeit für die Bearbeitung der Schleife hauptsächlich von der Menge des freien Speichers abhängt, der für die Arbeitsdaten benötigt wird, d. h. die Speicherbandbreite oder -latenz bremst die Ausführung stärker als die Rechenleistung. Compute-Bound hingegen bedeutet, dass die Algorithmen in der Partikelschleife und somit die Rechenleistung der entscheidende Faktor sind. Memory-Bound könnte in der Partikelschleife dabei u. a. durch das ständige hin- und herkopieren der Teilchenstrukturen durch die Funktion `memcpy` passieren. Benchmark-Tests sowie eine Analyse mittels des VTune Profilers und des Memory-Access-Tools zeigten jedoch, dass die Partikelschleife stark compute-Bound ist und lediglich einen geringen Speicher in Anspruch nimmt. Ein weiteres Ergebnis des Memory-Access-Tools war, dass der **Großteil des allozierten Speichers im Cache** lag, welcher sich auf den schnellsten L1-Cache und dem langsameren L3-Cache

verteilte. L1-Cache ist dabei relevant für die einzelnen Threads, während der L3-Cache als gemeinsamer Speicher für alle Threads dient, da er eine größere, aber langsamere Speicherebene ist als L1 und L2. Aufgrund der Nutzung des Cache anstelle des Arbeitsspeichers (RAM) wurde die Partikelschleife somit auch effizient optimiert hinsichtlich der Nutzung des Speichers.

Ebenso war zu kontrollieren, ob eine **möglichst ausgeglichene Auslastung der Threads** vorliegt. Ist die Auslastung auf die Threads ungleich verteilt (Load imbalance), kann es passieren, dass am Ende der Partikelschleife einige Threads nur noch warten, während andere Threads noch am Arbeiten sind. Dies ist keine optimale Situation, da die wartenden Threads theoretisch noch Arbeiten könnten. Hier kann das sogenannte Scheduling der Threads helfen, d. h. das Vorgehen, wie die Iterationen in einer parallelen Umgebung aufgeteilt werden. Der Standard ist die Option `static`. Bei dieser Option werden die Arbeiten in vordefinierten Blöcken (sogenannte Chunks, variierbar über Parameter) den Threads fest zugeteilt. Diese statische Methode ist insbesondere dann effektiv, wenn jedes Teilchen in der Schleife eine ähnliche Arbeitslast aufweist. Eine weitere Methode ist die Option `dynamic`. Wie der Name schon sagt, werden die Arbeiten auf die Threads dynamisch verteilt, d. h. wenn ein Thread seine Iteration beendet hat, wird ihm direkt eine neue Iteration zugewiesen. Diese Methode sollte gewählt werden, wenn die einzelnen Iterationen in ihrer Arbeitslast unterschiedlich stark ausfallen. Nachteil der Methode ist, dass durch die ständige dynamische Aufteilung ein Overhead entsteht. Ein Kompromiss zwischen `static` und `dynamic` ist die Option `guided`. Sie ähnelt der dynamischen Method und startet mit relativ großen Blöcken, welche mit fortschreitender Iterationszahl abnehmen. Ziel dabei ist, dass alle Threads durchgehend ausgelastet sind, aber dennoch die Häufigkeit der dynamischen Aufteilungen geringer ausfallen als bei der dynamischen Methode. Dadurch wird eine gute Balance für gemischte Arbeitslasten erreicht. In ARTM sollte die Methode `guided` die beste Wahl sein. Während die in jedem Zeitschritt neu erzeugten Teilchen alle eine ähnliche Arbeitslast aufweisen, können die schon aus dem vorherigen Zeitschritt in den nächsten Zeitschritt propagierten Teilchen eine stark variierende Arbeitslast verursachen. Ein veranschaulichtes Beispiel ist in Abb. 6.1 gegeben.

Für jeden Thread wurde die Zeit gemessen, wie lange er für seine Arbeit in der parallelen Umgebung braucht. Die Variation lag dabei bei unterhalb von 1 %, weswegen eine starke Load imbalance ausgeschlossen werden kann. Um sicherzustellen, dass eine mögliche Load imbalance gehandhabt werden kann, wurde als Schedule `guided` ausgewählt.

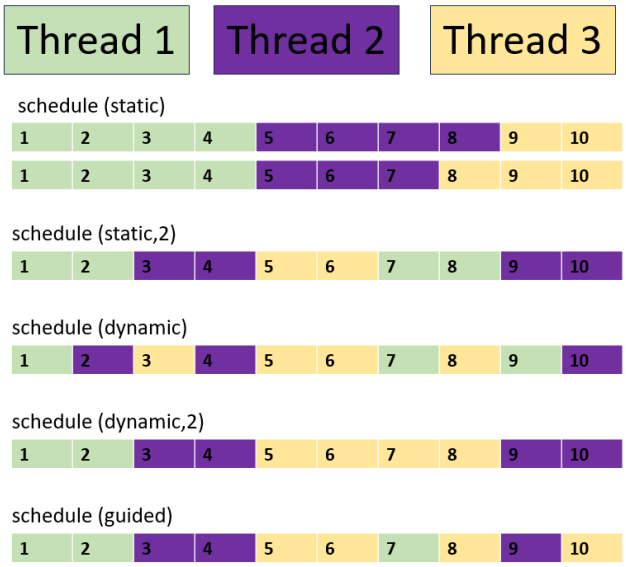


Abb. 6.1 Anschauliche Erklärung der unterschiedlichen OMP-Scheduling Methoden anhand von drei Threads und zehn Iterationen

Die Abbildung zeigt die Zuteilung der einzelnen Iterationen zu den jeweiligen Threads. Jeder Thread hat dabei seine eigene Farbe, damit die Einteilung gemäß der unterschiedlichen OMP-Scheduling Methoden anschaulich ist.

6.1.2 Vermeidung von False-Sharing

Zu Beginn einer parallelen Umgebung können OpenMP Direktiven für jeden Thread eine private Kopie einer globalen Variable erstellen. Eine weitere Möglichkeit, die z. B. zur Erzeugung der zufälligen Zahlen für den turbulenten Transport der Teilchen verwendet wurde, ist die Anpassung der Variablen zu Arrays. Durch Übergabe der entsprechenden Thread-Nummer in der parallelen Umgebung ändert somit jeder Thread seine eigene globale Variable innerhalb des Arrays. Bei dem Wechsel zu den Arrays war aufgefallen, dass die Programmlaufzeit von ARTM im sequenziellen und parallelen Ablauf zunächst identisch war, was nach einer längeren Recherche auf ein „False Sharing“ Problem zurückzuführen war. Prozessoren nutzen mehrstufige Cache-Ebenen, die unterschiedlich groß und auch schnell sind. Im L1-Cache werden die am häufigsten benötigten Daten zwischengespeichert, damit möglichst wenige Zugriffe auf den deutlich langsameren RAM (Arbeitsspeicher) notwendig sind. Moderne Prozessorarchitekturen arbeiten nicht mehr mit einzelnen zwischengespeicherten Werten, sondern nutzen sogenannte Cache-Lines. Dabei werden neben den explizit angeforderten Daten auch benachbarte Speicherbereiche gemeinsam in die Cache-Line geladen, um die Speicherzugriffszeiten durch räumliche Lokalität zu optimieren. Wenn nun jedoch mehrere Prozessorkerne mit jeweils einem Thread denselben oder benachbarte Speicherbereiche nutzen, kann es

passieren, dass sie sich dieselbe Cache-Line teilen. Sobald ein Prozessorkern eine Variable in seiner Cache-Ebene ändern will, muss der andere Prozessorkern darauf warten, dass dieser fertig ist und selbst seine Variable ändern kann. Eine Darstellung dieses Zusammenhangs ist in Abb. 6.2 gegeben.

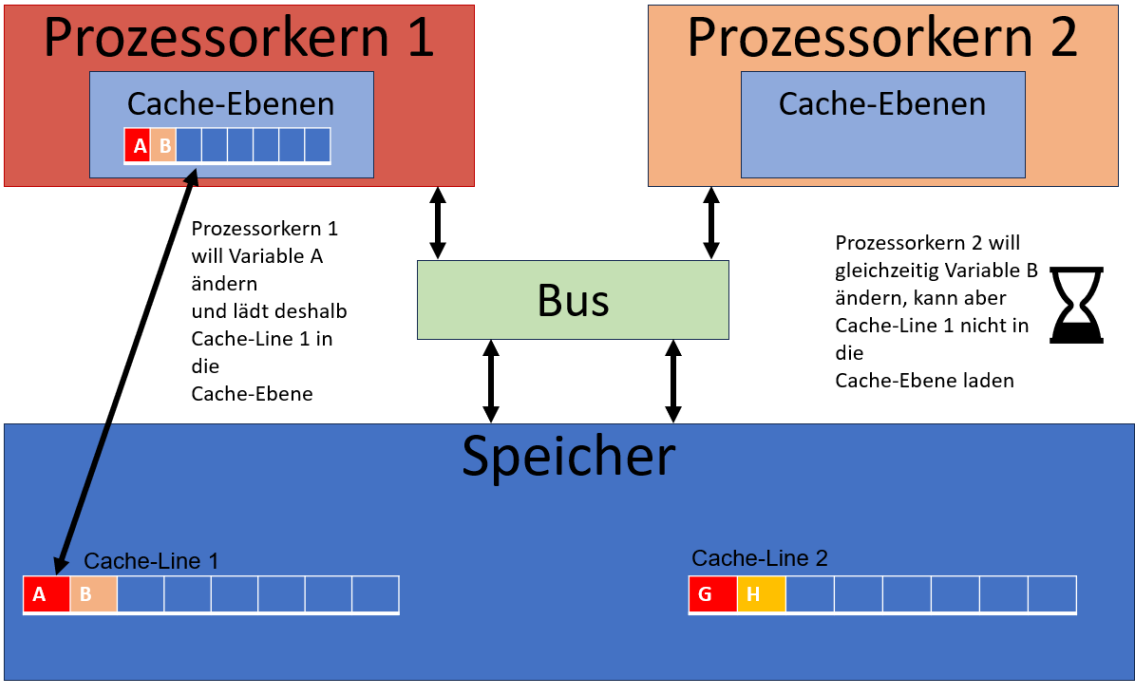


Abb. 6.2 Veranschaulichung eines False Sharing zweier Prozessorkerne

In einer Cache-Line befinden sich beispielsweise zwei Variablen A und B. Prozessorkern 1 will nun Variable A ändern und lädt die Cache-Line in seine Cache-Ebene. Gleichzeitig will Prozessorkern 2 die Variable B ändern. Da diese jedoch in derselben Cache-Line liegen, muss der Prozessorkern 2 darauf warten, dass der Prozessorkern 1 die Cache-Line wieder freigibt. Die Cache-Lines müssen somit ständig zwischen den Kernen (bzw. den Threads) synchronisiert werden, was die Leistung des Programmes massiv einschränken kann.

Heutzutage werden meistens 64-Bit Systeme verwendet, eine Cache-Line ist somit 64 Bytes groß. Beispielsweise würde ein Array aus acht Doubles (8 x 8 = 64 Bytes) in solch eine Cache-Line passen. Würde global solch ein Array nun definiert werden und jeder Prozessorkern würde auf dieses Array zugreifen wollen, würden die ersten acht Prozessorkerne in ein „False Sharing“ hineinlaufen. Um dieses Problem zu umgehen, werden die Daten „gepadded“. Padding bedeutet dabei, dass ein entsprechender Abstandshalter eingefügt wird, so dass jeder Thread seinen Wert des Arrays in einer anderen Cache-Line hat.

Ein typisches Beispiel für die Optimierung einer Double-Variable wäre:

```
#define CACHELINE 64
typedef struct {
    double value;
    char pad[CACHELINE - sizeof(double)];
} PaddedDouble;
```

Eine elegantere Lösung ist jedoch, interne Funktionen zu nutzen, die eine direkte Ausrichtung auf 64 Byte erzielen und die Größe der Struktur auf ein Vielfaches von 64 Byte aufrunden. Für das obige Beispiel wäre dies:

```
#if defined(_MSC_VER) || defined(__INTEL_COMPILER)
    #define ALIGN64 __declspec(align(64))
#else
    #define ALIGN64 __attribute__((aligned(64)))
#endif

typedef struct {
    double value;
} PaddedDouble ALIGN64;
```

Der Vorteil dieser Methodik ist, dass kein „False Sharing“ mehr möglich ist, da jede PaddedDouble Variable seine eigene Cache-Line erhält. Nachteil ist der höhere Speicherverbrauch. Dieser führte bei den Tests von ARTM mit dem Benchmark jedoch zu keinen Problemen. Alternativ kann auch das Array der Doubles künstlich vergrößert werden. Sollen beispielsweise lediglich zwei Threads parallelisiert werden, so müsste das Array der Doubles anstelle von `A[2]` auf `A[16]` erweitert werden. Für Floats, die lediglich 4 Byte groß sind, wäre das analoge `A[32]`. Der Zugriff muss in diesem Fall auf das Element `A[0]` und `A[8]` bei Doubles geschehen, bei Floats auf `A[0]` und `A[16]`.

Neben den Standardgrößen Double, Float und Int wurden ebenfalls viele der in ARTM erzeugten Strukturen passend „gepadded“. Dabei wurde sich zunächst auf die in der parallelen Schleifenebene notwendigen Strukturen beschränkt.

Im Rahmen der Änderung der Strukturen wurde ebenfalls entdeckt, dass die Variablen innerhalb der Strukturen teils sehr willkürlich deklariert werden. Zielführender ist jedoch, dass die Unterteilung in sogenannte **„Hot/Cold“-Fields** geschieht. Dabei werden häufig genutzte Variable („hot fields“) in den ersten 64 Bytes eingeordnet, während seltener genutzte Variablen („cold fields“) in dem restlichen Speicher der Struktur landen. Diese Umverteilung wurde beispielsweise in der Struktur `MD3REC` realisiert, welche Informationen über das Modellfeld beinhaltet.

6.1.3 Vermeidung von Race-Bedingungen

Eine Race-Bedingung ist eine Situation, die unbedingt bei der Parallelisierung vermieden werden muss. Sie trifft auf, wenn das Programm durch mehrere parallele Abläufe, gleichzeitig auf dieselben Daten zugreifen will, der richtige Programmablauf dies aber nicht erlaubt und eine sequenzielle Abarbeitung verlangt. Bei ARTM ist dies beispielsweise der Fall beim Zugriff und Schreiben in das Array, welches die Konzentrationswerte innerhalb der Gitterzellen erfasst. Da das Array eine globale Größe ist, kann das Lesen und Schreiben in ein Element des Arrays fast zur gleichen Zeit von unterschiedlichen Threads eingehen. Ein simples Beispiel einer Race-Bedingung wäre wie folgt:

1. Eine globale Variable `a` wird auf den Wert 1 initialisiert.

2. Es werden zwei parallele Threads erstellt, die beide nun diese globale Variable um 1 erhöhen sollen.

3. Abhängig vom zeitlichen Verlauf kann nun folgendes passieren.

    a) Thread 1 liest die globale Variable `a` ein. Der Wert ist zu diesem Zeitpunkt 1. Anschließend erhöht Thread 1 den Wert um 1 und speichert ihn. Der gespeicherte Wert ist nun 2. Denselben Ablauf führt nun Thread 2 durch. Der gespeicherte Wert am Ende ist 3. Dies ist das gewünschte Verhalten.

    b) Thread 1 und Thread 2 handeln komplett zeitgleich, da sie nicht miteinander kommunizieren. Thread 1 und Thread 2 lesen gleichzeitig die globale Variable `a` ein, erhöhen gleichzeitig die globale Variable um 1 und letztendlich speichern sie den Wert. Der gespeicherte Wert ist am Ende 2, obwohl 3 gewünscht ist.

Zur Vermeidung des Problems muss daher der erste Thread den Zugriff auf den Wert bis zum Abschluss der Veränderung so blockieren, dass Thread 2 warten muss, bis Thread 2 seinen Zugriff auf den Wert beendet hat. Diese Synchronisierung ist mit den OpenMP-Direktiven `omp atomic` und `omp critical` möglich. Eine von mehreren Anpassungen in ARTM war beispielsweise deshalb:

```
        if (washout && l.i>0 && l.j>0) {// Washout
          k = -1;
          pf = AryPtr(Pda, l.i, l.j, k, cmpoff);  if (!pf)eX(10);
          if (Pwo && rain2dinfo>0) {//falls 2D-Regen aktiviert und Info
vorliegt
              pw = AryPtr(Pwo, l.i, l.j, cmpoff);  if (!pw)eX(21);
          }
          for (icmp=0; icmp<cmpnum; icmp++) {
               //falls 2D-Regen infos da,
            if (Pwo && rain2dinfo>0)rwsh = pw[icmp];
            else rwsh = pr[icmp].rwsh; //ansonsten aus AKTERM bzw. zeitrei-
hen.dmna
            dg = e.g[2*icmp]*rwsh*tau;
            e.g[2*icmp] -= dg;
             #pragma omp atomic
            pf[icmp]+= dg;}
```

In diesem Code-Ausschnitt wird in die einzelnen Gitterelemente (`pf`) des Konzentrationsfeldes (`Pda`) für die nasse Deposition (dritte Dimension `k=-1`) geschrieben. `dg` ist dabei die deponierte Masse des Teilchens `e`. Die Direktive bewirkt dabei, dass, so lange einer der Threads auf ein Gitterelement zugreift, andere Threads, die ebenfalls auf genau dieses Gitterelement zugreifen wollen, warten müssen. Beim Teilchenzugriff muss keine Direktive gesetzt werden, da diese privat für jeden einzelnen Thread existiert und nicht wie die andere geteilt wird. Eine Möglichkeit ebenfalls für das Konzentrationsfeld die Direktiven zu vermeiden war deshalb eine Kopie für jeden Thread zu ermöglichen. Diese Variante wurde ebenfalls getestet. Aufgrund der großen Dimension des Konzentrationsfeldes war die Erstellung der Kopien zeitintensiver als das simple Setzen der OMP-Präprozessor Direktive (Laufzeit in Summe 219 s zu 236 s). Ein Teilausschnitt einer Analyse mittels des Leistungsprofilers Intel VTune Profilers /INT 25c/ für die beiden Methoden ist in Abb. 6.3 gezeigt. Die Methode mit kopierten Konzentrationsfeldes für jeden Thread hat zwar eine höhere maximale Parallelität (vgl. Histogramm in Abb. 6.3), dafür ist der serielle Anteil (18 s zu 44 s) zum Erstellen der Konzentrationsfeldes deutlich höher. Bei der Methode, in der der Zugriff auf die Konzentrationsfeldes durch atomic Bedingungen kontrolliert werden, ist der parallele Anteil (201 s zu 192 s) größer, was aufgrund der Synchronisierung durch die atomic Bedingung auch zu erwarten war. Es wird vermutet, dass mit komplexeren Konzentrationsfeldes (feineres Gitter) die Erstellung der Kopien mehr Rechendauer brauchen als die atomic Bedingungen in der parallelen Umgebung. Daher wurde sich für das Setzen der atomic Bedingung entschieden.

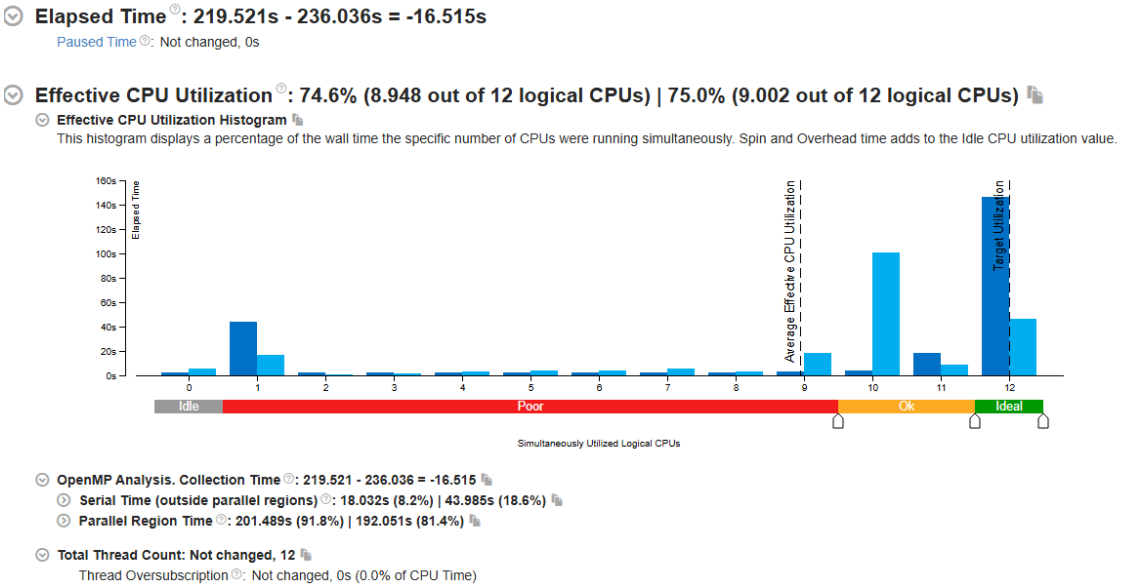


Abb. 6.3 Teilausschnitt des Intel VTune Profilers für den Vergleich der Methode, in der der Zugriff auf die Konzentrationsfeldes durch atomic Bedingungen kontrolliert wird, zur Methode mit kopierten Konzentrationsfeldes für jeden Thread

Der Graph zeigt die Zeit, wie viele CPUs parallel arbeiten. Hellblau ist dabei die Methode durch atomic Bedingungen, während dunkelblau die Methode der kopierten Konzentrationsfelder darstellt. Die linken Zahlenwerte gelten dabei für die atomic Bedingung, die rechten Zahlenwerte für die kopierten Konzentrationsfelder.

Ein weiteres Beispiel für eine Race-Condition war in dem Zugriff auf die Partikel selbst gegeben. Der Code, um innerhalb des Arrays aus Teilchen auf das nächste Teilchen zuzugreifen war, wie folgt:

```
void *PtlNext(  /* pointer to next ptl  */
  int handle)  /* handle               */
  {
  dP(PtlNext);
  void *pp;
  ARYDSC *pa;
  struct ARRLIST *ps;
[…]
  while (ps->i<=ps->n) {
    ps->i++;
    pp = AryPtr(pa, ps->i-1);  if (!pp)              eX(1);
    if (*(PTLTAG*)pp)  return pp;
  }
[…]
```

In dieser Form konnte es passieren, dass `ps->i++` durch den gleichzeitigen Zugriff falsch erhöht wird, und jeder Thread nicht sicher auf unterschiedliche Teilchen zugreifen

kann. Aus diesem Grund musste sowohl die for-Schleife in der Teilchenpropagation als auch die Funktion `PtlNext()` angepasst werden:

```
void *PtlNext(  /* pointer to next ptl  */
  int handle, /* handle               */
  int i) /*Teilchenindex im Array*/
  {
  dP(PtlNext);
[…]
  while (i<=ps->n) {
    if (i==0)i++;
    pp = AryPtr(pa, i);  if (!pp)              eX(1);
    if (*(PTLTAG*)pp)  return pp;
  }
[…]
```

Anstelle einen Laufindex zu erhöhen, wurde nun in der Funktion direkt der Index eines Partikels innerhalb des Arrays übergeben. Da nun auch keine Race-Bedingung mehr existiert, konnte somit eine Synchronisierung der Threads vermieden werden.

6.2 Umsetzung der Parallelisierung von TALdia

Um die Parallelisierung in TALdia gemäß dem erarbeiteten Konzept umzusetzen, wurde zunächst die formale Voraussetzung für die Anwendung von OpenMP geschaffen. Das beinhaltete neben der Einbindung der benötigten Header-Datei „omp.h“ die Übersetzung einer `while`-Schleife in eine `for`-Schleife. Im Anschluss daran wurde diese Schleife mit OpenMP spezifischen Präprozessor Direktiven parallelisiert. Diese Schleife befindet sich unmittelbar in der main-Funktion von TALdia und damit in der Hierarchie des Aufrufstapels sehr weit unten. In dieser Schleife wird dann eine Vielzahl von Funktionen aufgerufen, die ihrerseits wiederum Funktionen aufrufen. Dadurch wächst der Aufrufstapel stark an. Die Parallelisierung der Schleife in einer Funktion, die sich sehr weit unten im Aufrufstapel befindet, wirkt sich auch stark auf die Funktionen aus, die sich weiter oben im Aufrufstapel befinden. So wurden Änderungen an einem großen Teil des Quellcodes nötig. Dieser Anpassungsbedarf lässt sich in zwei Hauptgruppen unterteilen, die Umwandlung von globalen Variablen in threadspezifische globale Variablen und die Synchronisierung von Programmabläufen. Darüber hinaus ergab sich durch die Parallelisierung die Notwendigkeit, die allgemeine Datenhaltung von TALdia bzw. ARTM, den sogenannten Table Manager, zu modifizieren. Auf diese drei genannten Themenbereiche wird im Folgenden näher eingegangen.

6.2.1 Umwandlung von globalen Variablen in threadspezifische globale Variablen

Wie ARTM selbst, arbeitet auch TALdia mit einer großen Zahl von globalen Variablen. In einem sequenziellen Programm wird eine globale Variable nur von einem Thread verarbeitet. Bei einem parallelen Programm arbeiten mehrere Threads parallel mit derselben globalen Variablen, was zu einer Race-Bedingung führt (vgl. Abschnitt 6.1.1 und Abschnitt 6.1.3). Daher wurde in TALdia eine große Zahl von globalen Variablen in Arrays von threadspezifischen Variablen umgewandelt. Die eindeutige Thread-Nummer entsprach dem Array-Index, so dass jeder Thread nur noch mit seinem ihm zugewiesenen Array-Element arbeitete.

6.2.2 Synchronisierung von Programmabläufen

Für die Parallelisierung von TALdia wurde es notwendig, einige Programmabläufe zeitlich zu koordinieren. Dies war überwiegend, aber nicht ausschließlich, bei lesenden und schreibenden Zugriffen auf der Festplatte der Fall. Der Programmablauf von TALdia sieht diese Festplattenoperationen innerhalb des zu parallelisierenden Bereichs des Programms vor. Parallele Festplattenoperationen können jedoch zu Konflikten führen und wurden daher programmiertechnisch ausgeschlossen. Ein Beispiel für eine parallele Festplattenoperation, die sich durch den Programmablauf von TALdia ergibt, ist das Einlesen von Grenzschichtparametern. Die Grenzschichtparameter werden in zwei Schritten aus der Datei `metlib.def` eingelesen. Dabei übernimmt zunächst die Funktion `BlmRead()` aus der Datei `TalBlm.c` das Einlesen des ersten, allgemeinen Teils der Parameter. Danach erfolgt in einem zweiten Schritt das Einlesen der meteorologischen Parameter durch die Funktion `ReadZtr()` aus der Datei `TalBlm.c`, die nur für das Windfeld relevant sind, das aktuell berechnet werden soll. Im Falle des sequenziellen Programmablaufs von TALdia, bedeutet das, dass für jedes Windfeld zweimal lesend auf die Datei `metlib.def` zugegriffen wird. Bei einem parallelen Programmablauf, so wie er konzeptioniert wurde, besteht die Gefahr von konkurrierenden Festplattenzugriffen. Parallel ablaufende Threads, von denen jeder zeitgleich ein Windfeld bearbeiten soll, könnten zeitgleich lesend auf die Datei `metlib.def` zugreifen, was verhindert werden sollte. Für dieses Problem konnten zwei Ansätze identifiziert werden.

Der erste Ansatz ist die zeitliche Steuerung des Zugriffs jedes Threads auf die gewünschte Datei. Eine schematische Darstellung der Abläufe ist in Abb. 6.4 abgebildet. Das Einlesen der allgemeinen Parameter wird nur einem Thread, dem Master-Thread,

erlaubt. Während dieses Vorgangs haben die anderen Threads (Arbeiter-Threads) keine Arbeiten zu erledigen und warten untätig darauf, dass der Master-Thread seine Arbeit beendet. Anschließend werden die eingelesenen Parameter jedem Arbeiter-Thread durch das Klonen der entsprechenden Variablen privat zur Verfügung gestellt. Hierbei ist jeder Arbeiter-Thread für die Erstellung seines eigenen Klons verantwortlich. Das Einlesen der meteorologischen Parameter, die spezifisch für jedes Windfeld sind, wird zeitlich gesteuert. Es wird immer nur einem Thread erlaubt, zeitgleich auf die Datei `met_lib.def` zuzugreifen. Dabei entsteht zwangsläufig eine „Warteschlange“ mit untätigen, wartenden Threads. Der Vorteil dieses Ansatzes ist, dass wenige Änderungen am seriellen Quellcode gemacht werden mussten. Abgesehen von den obligatorischen Präprozessor Direktiven von OpenMP, musste nur noch eine Funktion entworfen werden, die das Klonen der allgemeinen Parameter ermöglichte.

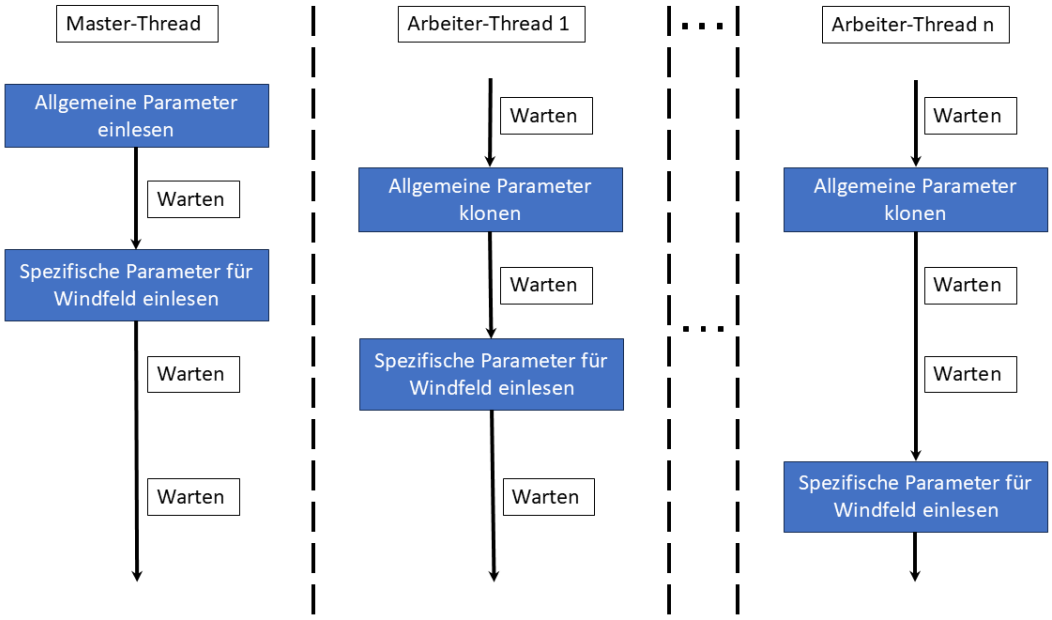


Abb. 6.4 Synchronisiertes Einlesen der Meteorologischen Parameter bei mehr als einem Thread

Der zweite Ansatz zum Einlesen von meteorologischen Parametern aus der Datei `met_lib.def` ist die Änderung des Programmablaufs. Anstatt die Datei bei jeder einzelnen Berechnung eines Windfeldes aufs Neue einzulesen, könnten die meteorologischen Parameter bereits vor dem parallelen Programmbereich eingelesen und in geeignete Datenstrukturen gespeichert werden. Diese sind allen späteren Threads über das „shared-memory“ System zugänglich. Somit wäre ein paralleler lesender Zugriff auf die meteorologischen Parameter ohne die Gefahr von Zugriffskonflikten möglich. Darüber hinaus gäbe es keine Wartezustände der Threads mehr, wie es beim ersten Ansatz der Fall ist.

Zur Realisierung des zweiten Ansatzes wäre eine Neuimplementierung der Einleseroutine für die `metlib.def` Datei, sowie das Entwerfen einer geeigneten Datenstruktur, nötig. Zudem würden Funktionen benötigt, die es jedem Thread ermöglichen, aus den eingelesenen Daten die Teilmenge zu identifizieren, die für die Berechnung seines Windfeldes benötigt wird.

6.2.3 Änderungen der allgemeinen Datenhaltung von ARTM für eine parallele Programmausführung

In ARTM, und damit auch in TALdia, werden nahezu alle Daten, die eingelesen oder im Programmverlauf berechnet werden, durch den Table Manager verwaltet und gespeichert. Der Table Manager ist eine Sammlung von Funktionen, die für die Allokation von Speicherplatz, die Initialisierung von Datensätzen, die Speicherung auf der Festplatte, das Einlesen von der Festplatte, das Durchsuchen sowie das Bereitstellen von Daten und das Löschen von Daten und Datensätzen aus dem Hauptspeicher verantwortlich sind. Die Datensätze selbst, sowie dazugehörige Metadaten, werden in separaten komplexen Strukturen abgelegt. Diese komplexen Strukturen sind hierarchisch so miteinander verbunden, dass sie eine doppelt verkettete Liste bilden. Jedes Listenelement kennt sein unmittelbar vorheriges und sein unmittelbar nachfolgendes Listenelement /WOL 19/, /KAN 22/.

Doppelt verkettete Listen sind im Allgemeinen nicht dafür ausgelegt, dass mehr als eine Operation zur selben Zeit auf ihnen ausgeführt wird. So können beispielsweise konkurrierende Operationen zum Löschen von zwei Listenelementen dazu führen, dass nur eines der Listenelemente tatsächlich gelöscht wird /VAL 95/. Auf eine genaue Diskussion, warum das so ist, wird an dieser Stelle verzichtet, da dies den Rahmen dieses Berichts sprengen würde. Um die Einschränkung von verketteten Listen für konkurrierende Operationen zu lösen, wurden über die Jahre verschiedene Verfahren entwickelt /HAK 04/, /GRE 02/, /HAR 01/, /VAL 95/.

Da es sich bei der verketteten Liste in TALdia aber nur um eine kurze Liste handelt (15 – 30 Listenelemente), wurde eine Methode verwendet, die weniger Implementierungsaufwand erforderte, als es bei der Implementierung eines der oben zitierten Verfahren der Fall gewesen wäre. Jeder Thread erhielt einen eigenen, privaten Klon der doppelt verketteten Liste in dem Zustand zum Zeitpunkt der Erstellung der parallelen Threads. So arbeitet jeder Thread auf seiner eigenen Datenstruktur und die Änderungen an den Funktionen des Table Managers konnten minimal gehalten werden. Um die

verkettete Liste zu klonen, wurden die Funktionen `clone_list()`, `clone()`, `clone_start_data()`, `get_private_head()`, `get_head()` und `init_tmn_head_clones()` in der Datei `TalTmn.c` neu implementiert. Für die zusätzlichen geklonten verketteten Listen wurde Speicherplatz im Hauptspeicher dynamisch alloziert. Um diesen Speicherplatz nach der Berechnung aller Windfelder wieder freizugeben, wurde die Funktion `clone_FREE()` implementiert.

Die Vorteile des threadspezifischen Klonens von Datenstrukturen bestehen darin, dass jeder Thread seine private Datenbasis zur Verfügung hat, auf der dieser Operationen durchführen kann. Auf diese Weise wird verhindert, dass es zu Wartezuständen der verschiedenen Threads bei Operationen auf der verketteten Liste kommt. Der Nachteil ist die Redundanz der Datenbasis und damit ein hoher Speicherplatzverbrauch während des Programmlaufs von TALdia. Der Speicherplatzbedarf skaliert linear mit der Anzahl der verwendeten Threads, der Anzahl der geschachtelten Gitter und der Anzahl der Gitterzellen in einem Simulationsgitter. Im betrachteten Benchmarktest belegte TALdia zu keinem Zeitpunkt mehr als ca. 60 MB Hauptspeicher. Komplexere Simulationen können den Speicherverbrauch allerdings steigen lassen.

6.3 Vergleich der sequenziell und parallel berechneten Simulationsergebnisse

Bei allen Änderungen an einer Simulationssoftware muss sichergestellt sein, dass die Änderung keinen unerwünschten Nebenwirkungen auf die Simulationsergebnisse hat. Daher müssen auch im Fall des Beispielprogramms ARTM die sequenziell und parallel berechneten Simulationsergebnisse übereinstimmen.

6.3.1 Vergleich der Ergebnisse der ARTM Simulationen

Um den Einfluss des umgesetzten Konzeptes zur Parallelisierung in ARTM auf der Ebene der Teilchenpropagation zu untersuchen, wurden alle berechneten Felder von einer parallelen Rechnung mit der dazugehörigen sequenziellen Rechnung verglichen. Als Test-Framework wurde pytest verwendet /KRE 24/ und das Standard-Benchmark wurde dabei mit einer einmonatigen meteorologische Zeitreihe und einer Qualitätsstufe der Freisetzungsrate von Partikeln von 4 angepasst. Für alle Verteilungen (Konzentrationsfelder, nasse und trockene Deposition sowie Gammasubmersion) wurden die Ergebnisse der sequenziellen und parallelen Rechnung mit den dazugehörigen Unsicherheiten (Standardabweichungen) in ein jeweiliges Array geladen. Für jedes Element wurde an-

schließend geprüft, ob sich die Wertebereiche innerhalb einer definierten Standardabweichung überlappen. Anschließend wurde zusätzlich die maximale Abweichung (in %) zwischen den Intervallen berechnet und ein Histogramm dieser Abweichungen erstellt. Die Tests für die nasse und trockene Deposition waren innerhalb der statistischen Unsicherheiten für alle Elemente erfolgreich. Bei Br82 kam es bei der Konzentration bei etwa 1 % der Werte zu einer maximalen Abweichung von weniger als 1 %. Bei allen anderen Elementen waren die Tests erfolgreich. Die Tests für die Gammasubmersion, für die keine statistischen Unsicherheiten vorlagen, sind fehlgeschlagen. Die maximale Abweichung lag aber bei unter 1 %. Die Ursache der Abweichungen, insbesondere bei der Gammasubmersion, ist noch nicht geklärt.

Die Maximalwerte und die Position der jeweiligen Konzentrationsfelder sind identisch. Ein Vergleich aller Ergebnisfelder zwischen paralleler und sequenzieller Berechnung selbst ergab optisch ebenfalls kein Unterschied.

6.3.2 Vergleich der Windfelder

Um den Einfluss des umgesetzten Konzepts zur Parallelisierung von TALdia auf die Windfelder zu prüfen, wurden die berechneten Windfelder im Textformat miteinander verglichen. Dabei wurde TALdia mit dem Kommandozeilenparameter `-t` dazu gebracht, die Windfelder nicht im Binärformat, sondern im Textformat auszugeben. Der Vergleich wurde ausschließlich anhand des Benchmark Projekts, allerdings ohne Gebäude, durchgeführt. Für den Vergleich wurde ein Python Skript geschrieben, das die Windfelder des sequenziell und parallel arbeitenden Programms miteinander vergleicht und, wenn vorhanden, deren Differenz berechnet. Beim Vergleich der Windfelder ergaben sich bei jeder Windfelddatei kleine Unterschiede. Bei diesen Unterschieden handelte es sich um unterschiedliche Windgeschwindigkeiten in den drei Raumrichtungen an der dritten Nachkommastelle. Die Häufigkeit dieser Abweichungen ist im Vergleich mit der hohen Anzahl der übereinstimmenden Windgeschwindigkeitswerte verschwindend gering (im Bereich von 0,002 %). In Abb. 6.5 ist beispielhaft ein Histogramm mit den Abweichungen

zwischen dem sequenziell und parallel berechneten Windfeld für den Fall einer sehr stabilen Ausbreitungsklasse mit einer Windrichtung aus 180 ° gezeigt.

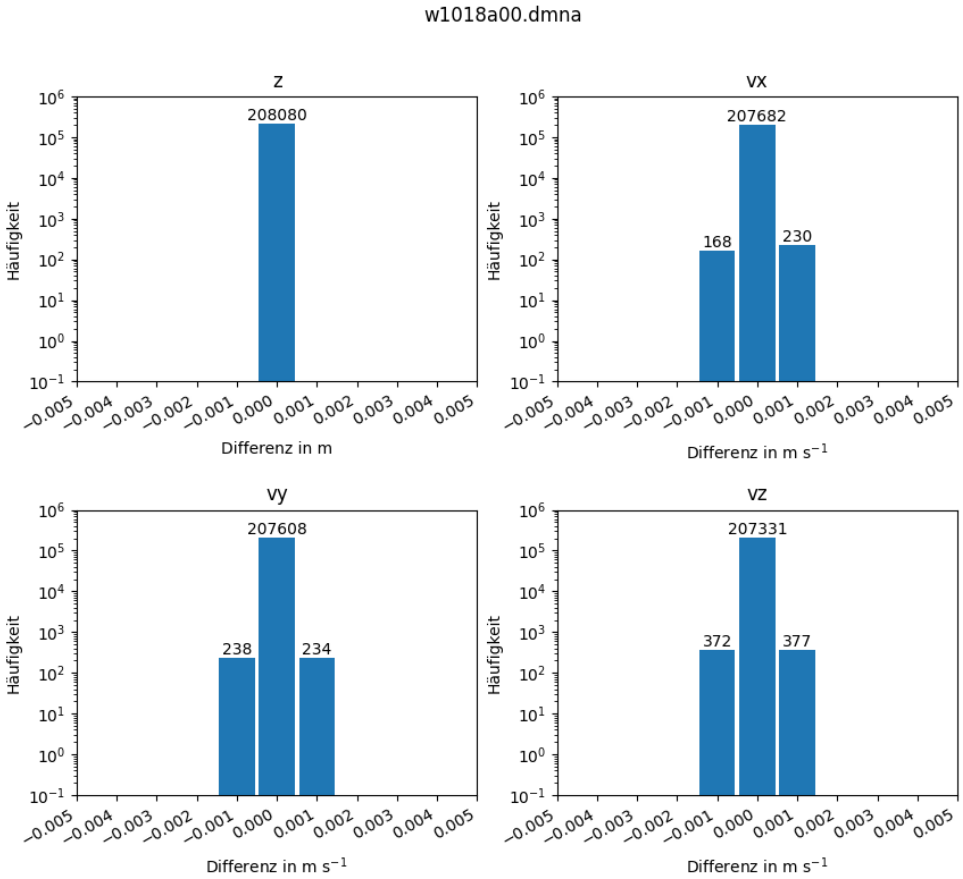


Abb. 6.5 Histogramm für die Differenzen zwischen dem sequenziell und parallel berechneten Windfeld für die sehr stabile Ausbreitungsklasse bei einer Windrichtung aus 180 °

Die Windfelddatei besteht aus 208.080 Datenpunkten. Für jeden dieser Datenpunkte ist ein Höhenwert z in m sowie die mittleren Windgeschwindigkeiten (vx, vy, vz) in m s^{-1} gespeichert. Die Höhenwerte z sind für das sequenziell und parallel berechnete Windfeld identisch. Bei den Windgeschwindigkeiten weichen einige Hundert Werte der sequenziell und parallel berechneten Windfelder um den Betrag 0,001 m s^{-1} voneinander ab. Abweichungen um größere Beträge treten nicht auf.

Wie sich herausstellte, sind die beobachteten Abweichungen allerdings nicht das Resultat der Parallelisierung der Windfeldberechnung, sondern ließen sich auf die verwendeten Kompilierer zurückführen. Die in Abb. 6.5 verglichenen Windfelddaten stammten einerseits von dem sequenziell arbeitenden Programm, das mit dem MSVC Kompilierer kompiliert wurde (ARTM/TALdia 3.1.3), während das parallel arbeitende Programm mit dem Intel C Kompilierer kompiliert wurde. Bei einem Vergleich zwischen den Ergebnissen eines sequenziell und eines parallel arbeitenden Programms, die beide mit dem

Intel C Kompilierer kompiliert wurden, ergaben sich keine Unterschiede. Darüberhinaus traten Unterschiede zwischen Windfeldern auch dann auf, wenn die Windfelder von zwei sequenziell arbeitenden Programmen verglichen wurden, von denen eines mit dem MSVC Kompilierer und eines mit dem Intel C Kompilierer kompiliert wurde. Es wird vermutet, dass die Ursache für die unterschiedlichen Windfelder für die beiden Kompilierer in der Implementierung des C Standards liegt. Der C Standard gibt die wichtigsten Regeln für mathematische Berechnungen zwar vor, regelt allerdings nicht die gesamte Implementierung. Da der Intel C Kompilierer eine eigene mathematische Bibliothek bereitstellt, kann es zwischen den Kompilierern zu unterschiedlichen Ergebnissen kommen, die im Fall von TALdia allerdings nicht signifikant sind.

6.4 Laufzeitverkürzung aufgrund der erarbeiteten Parallelisierungsmaßnahmen

Nach der Umsetzung des in Abschnitt 5.5 vorgestellten Konzepts konnten lauffähige Programme kompiliert werden. Allerdings befinden sich diese in einem experimentellen Stadium. Weite Teile der Programme TALdia bzw. ARTM müssten für eine abschließende Bewertung überarbeitet, angepasst und optimiert werden. Eine Liste des zum jetzigen Zeitpunkt prognostizierten Anpassungsbedarfs ist im folgenden Anschnitt 6.5 aufgelistet. Dennoch soll an dieser Stelle das grundsätzliche Ergebnis der nachträglichen Parallelisierung eines komplexen Programms am Beispiel von ARTM diskutiert und bewertet werden.

6.4.1 Laufzeitverkürzung durch Parallelisierung des Transportmodells von ARTM

Für die Parallelisierung der Partikelschleife in ARTM wurden am Ende der Optimierung mehrere Durchläufe des Intel VTune Profilers /INT 25c/ für das Modul „Threading“ durchgeführt. Der Vergleich wurde dabei auf einem Computer mit einem Intel® Core™ i7-1365U der 13. Generation und 1,80 GHz Taktfrequenz durchgeführt. Das Ergebnis des Benchmarks ist in Abb. 6.6 zu sehen. Der Graph zeigt, dass ein Großteil der Zeit lediglich ein Thread arbeitet, während nur vereinzelnd mehr Threads genutzt werden. Im Mittel wird eine CPU-Nutzung von nur etwa 36,5 % erreicht. Dennoch ist die parallele Version mit 74 s um einen Faktor 3,6 schneller als der sequenzielle Durchlauf des Programms mit einer Zeit von 267 s.

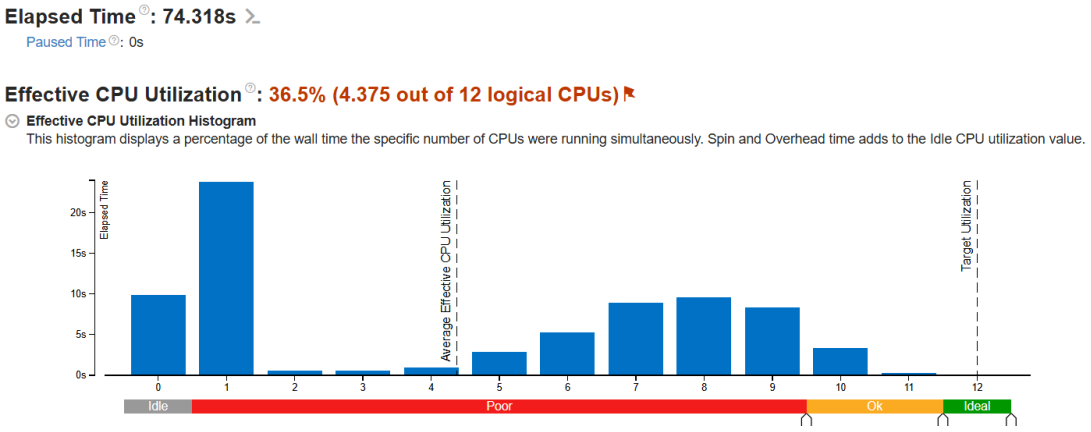


Abb. 6.6 Teilausschnitt des VTune Profilers für den betrachteten Benchmark aus Kapitel 5

Der Graph zeigt die Zeit, wie viele CPUs parallel arbeiten. Im folgenden Fall ist ein Großteil der Arbeit seriell durchgeführt. Die Rechnungen wurden auf einem Computer mit einem Intel® Core™ i7 1365U der 13. Generation und 1,80 GHz Taktfrequenz unter Windows ausgeführt.

Die schlechte Ausnutzung der CPU und somit der parallelen Umgebung ist damit zu erklären, dass bei einer Qualitätsstufe von 0 pro aufgebaute parallele Umgebung nur 1600 Teilchen im Benchmark propagiert wurden. Hier ist der Overhead im Vergleich zu der Arbeit der parallelen Umgebung für die Teilchenpropagation groß. Die CPU-Nutzung ändert sich direkt, wenn die Qualitätsstufe auf 4 erhöht wird. Dadurch werden 2^4 Teilchen mehr propagiert (25.600 Teilchen). Die Arbeit in der parallelen Umgebung ist somit größer als der Overhead. Ein Beispielergebnis des Benchmarks, welches auf eine einwöchige meteorologische Zeitreihe reduziert wurde und mit einer Qualitätsstufe der Freisetzungsrate von Partikeln von 4 berechnet wurde, ist in Abb. 6.7 zu sehen. Die Rechendauer betrug dabei etwa 194 s und die effektive CPU-Ausnutzung lag bei 82,5 %. Der sequenzielle Anteil betrug etwa 19 s, während der parallele Teil 175 s in Anspruch nahm. Das sequenzielle ARTM Programm ohne Parallelisierung brauchte für dasselbe Benchmark 736 s. Es wurde somit ebenfalls eine Beschleunigung von etwa einem Faktor 3,8 erreicht. Das komplette Benchmark mit einer einmonatigen meteorologischen Zeitreihe und einer Qualitätsstufe der Freisetzungsrate von Partikeln von 4 ergab eine Rechenzeit von 665 s. Eine vergleichbare Rechnung des sequenziellen Quellcodes ergab 2743 s, was eine Beschleunigung um einen Faktor 4 bedeutet.

Zu beachten ist, dass je nach Auslastung des Computers größere Schwankungen der Programmlaufzeit auftraten. Dies hing insbesondere davon ab, wie lange der Computer schon angeschaltet war und wie viele Prozesse im Hintergrund liefen. Aus diesem Grund

wurde das Profiling stets direkt nach dem Start des Rechners durchgeführt. Dadurch reduzierte sich eine anfängliche Variation der Laufzeit auf wenige Sekunden.

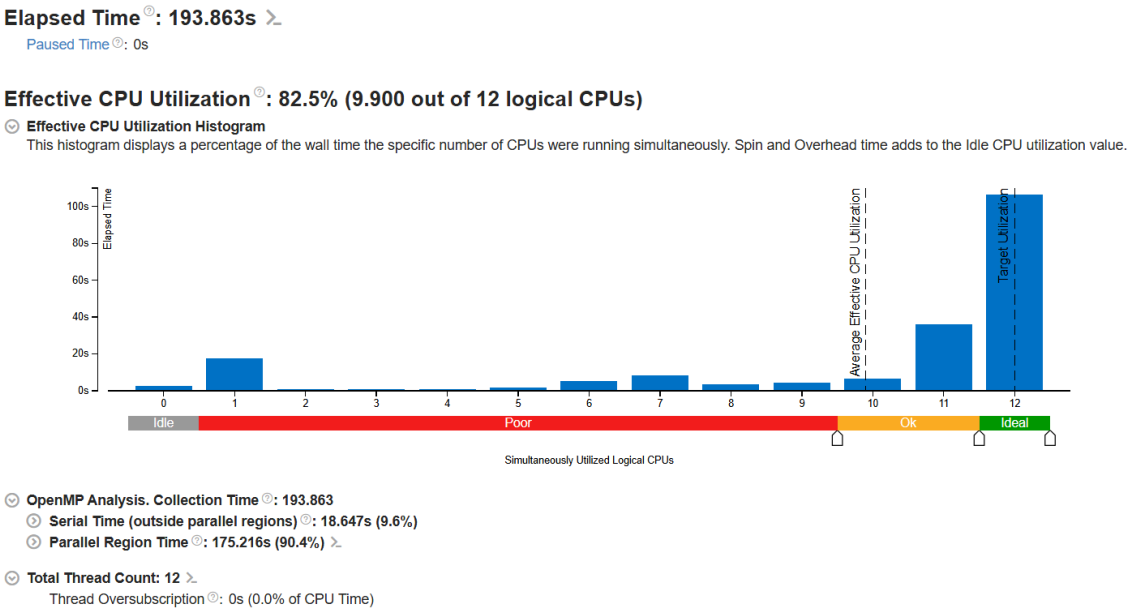


Abb. 6.7 Teilausschnitt des VTune Profilers für den angepassten Benchmark mit einer höheren Anzahl an zu propagierenden Teilchen

Der Graph zeigt die Zeit, wie viele CPUs parallel arbeiten. Im folgenden Fall ist ein Großteil der Arbeit parallel durchgeführt mit einer durchschnittlichen CPU-Nutzung von 9.9 CPUs, welche parallel rechnen. Ein Großteil der Zeit werden in der parallelen Umgebung alle 12 zur Verfügung gestellten CPUs genutzt. Die Rechnungen wurden auf einem Computer mit einem Intel® Core™ i7-1365U der 13. Generation und 1,80 GHz Taktfrequenz unter Windows ausgeführt.

Zusammenfassend wurde eine Beschleunigung des Programmes um mehr als einen Faktor 3 erreicht. Die Ergebnisse der unterschiedlichen Benchmarks und der Laufzeiten sind in Tab. 6.1 gegeben. Der theoretisch erzielbare maximale Beschleunigungsfaktor nach dem Amdahlschen Gesetz beträgt 6. Ein detailliertes Profiling konnte die Abweichung noch nicht erklären. Einen kleinen Anteil könnte die Tatsache haben, dass die Beschleunigung gemäß des Amdahlschen Gesetztes mit zwölf Kernen berechnet wurde und dabei von einer perfekten Skalierung ausgegangen wird. Tatsächlich sind jedoch nur zehn physische Kerne vorliegend, während zwei Kerne Simultaneous Multithreading (SMT)/Hyperthreading ermöglichen. Dies bedeutet, dass zwei der SMT/Hyperthreading Prozesse einen weiteren virtuellen Thread erstellen können, die sich die Ressourcen des physischen Kerns teilen. Während beispielsweise einer der Threads aufgrund von Warten auf Daten nicht arbeiten kann, kann der andere Thread die ALU nutzen, um Rechnungen durchzuführen. Bei Programmen, die sehr viel die ALU nutzen, kann somit

die Anzahl der tatsächlich nutzbaren Threads künstlich reduziert sein. Die perfekte Skalierung gemäß des Amdahlschen Gesetzes mit lediglich zehn Kernen läge bei 5,5, während sie mit 12 Kernen bei 6,0 liegt. Ferner ist zu beachten, dass in dem Amdahlschen Gesetz nicht beachtet wird, dass die parallele Umgebung aufgebaut werden muss. Dieser Overhead kann die perfekte Skalierung somit ebenfalls einschränken.

Tab. 6.1 Laufzeitvergleich für die sequenzielle und parallele Ausführung von ARTM mit entsprechenden Beschleunigungsfaktoren

Zu sehen sind die Laufzeiten der sequenziellen und parallelen Programmausführungen für die drei unterschiedlichen Benchmarks. Die dritte Spalte zeigt die Beschleunigung durch den Wechsel von sequenzieller zu paralleler Ausführung. Die Rechnungen wurden auf einem Computer mit einem Intel® Core™ i7-1365U der 13. Generation und 1,80 GHz Taktfrequenz unter Windows ausgeführt.

Programmablauf	sequenziell [s]	parallel [s]	Beschleunigungsfaktor
Benchmark (qs=0, 1 Monat AK-TERM)	267	74	3,6
Benchmark (qs=4, 1 Woche AK-TERM)	736	194	3,8
Benchmark (qs=4, 1 Monat AK-TERM)	2743	665	4,1

Erste Hinweise des Intel VTune Profilings mittels des Moduls „Microarchitecture Exposure“ deuten darauf hin, dass ebenfalls die Mikroarchitektur der Chips untersucht werden kann und ein Grund für die nicht perfekten Skalierung sein kann. Die starken Kerne im Chip (sogenannte P-Cores, kurz für Performance Cores /INT 25d/) haben ein „Retiring“ von 72 %. Das „Retiring“ ist dabei der Anteil der Takte, in denen die CPU nützliche Instruktionen fertig ausgeführt hat /INT 25a/. Ein Retiring von 72 % bedeutet, dass in 72 % der Takte die CPU tatsächlich Arbeit erledigt hat, während die restlichen 28 % „Stalls“[14] oder Overheads sind. Ein Wert über 70 % ist dabei schon sehr gut. Das Retiring der schwächeren Kerne (sogenannte E-Cores, kurz für Efficient Cores /INT 25d/) liegt hingegen lediglich bei 47 %. Weitere Größen müssten genauer für die E-Cores untersucht werden. P-Cores sind dabei leistungsstark und haben einen hohen IPC-Wert (Instruktionen pro Takt), während E-Cores kleiner und energieeffizienter sind und kleinere IPC-

[14] Moderne CPUs führen Befehle nicht strikt nacheinander, sondern in einer Pipeline und oft nicht in der vorgegebenen Reihenfolge aus. Im Idealfall ist die Pipeline voll und jede Taktung erledigt nützliche Arbeit. Dies führt zu einem hohen Retiring. Ein Stall tritt auf, wenn die Pipeline wartet, weil eine Ressource fehlt oder irgendeine Abhängigkeit nicht erfüllt ist. In solchen Fällen taktet der Kern, aber macht keine Arbeit.

Werte besitzen. Inwieweit die geringere IPC-Werte der E-Cores die Gesamtleistung des Programms beeinflusst, müsste dadurch untersucht werden, dass dem Programm lediglich die P-Cores zum Rechnen zur Verfügung gestellt werden. Hierbei zeigt sich erneut die starke hardwarespezifische Komponente der Parallelisierung.

6.4.2 Laufzeitverkürzung durch Parallelisierung von TALdia

In TALdia wurde die Schleife über die Berechnung der verschiedenen Windfelder parallelisiert. Da sich diese Schleife direkt in der main-Datei von TALdia befindet, führte die Parallelisierung zu einem großen Anpassungsbedarf von in den folgenden aufgerufenen Funktionen. Aufgrund der begrenzten Bearbeitungszeit konnten leider nicht alle Funktionen und damit Funktionalitäten von TALdia entsprechend angepasst werden. Beispielsweise wurden die Funktionen zur Handhabung von Gebäuden und geschachtelten Gittern bei der Parallelisierung nicht bearbeitet. Daher kann die Laufzeitverkürzung nicht, wie ursprünglich geplant, mit dem Benchmark-Projekt (vgl. Abschnitt 5.3) durchgeführt werden. Für den Laufzeitvergleich wird das Benchmark-Projekt ohne Gebäude verwendet. Der theoretisch erzielbare maximale Beschleunigungsfaktor nach dem Amdahlschen Gesetz ändert sich zu:

$$\eta_{B,\mathrm{TALdia}} = \frac{t_s + t_p}{t_s + \frac{t_p}{n_p}} = \frac{0{,}12 + 99{,}88}{0{,}12 + \frac{99{,}88}{12}} \approx 11{,}84 \tag{6.1}$$

Der Vergleich wurde auf einem Computer mit einem Intel® Core™ i7-1365U der 13. Generation und 1,80 GHz Taktfrequenz durchgeführt. Für den Vergleich wurden die Mittelwerte der Programmlaufzeiten aus mindestens zwei und maximal drei Programmdurchläufen unter Windows betrachtet. Die Mittelwertbildung wurde nötig, da während des Vorhabens festgestellt wurde, das die Programmlaufzeiten bei mehrfacher Ausführung über den Tag verteilt stark schwanken. Dieser Effekt trat sowohl bei der sequenziellen als auch bei der parallelen Version von TALdia auf. Der Grund dafür wird in der Ausführung von anderen Programmen und Hintergrundprozessen vermutet, die unter Windows auftreten, und der damit verbundenen Ressourcenbelegung. Die Mittelwerte der Programmlaufzeiten sind in Tab. 6.2 zusammengefasst.

Tab. 6.2 Laufzeitvergleich für die sequenzielle und parallele Ausführung von TALdia

Zu sehen sind die Mittelwerte aus drei sequenziellen und zwei parallelen Programmausführungen. Die Rechnungen wurden auf einem Computer mit einem Intel® Core™ i7-1365U der 13. Generation und 1,80 GHz Taktfrequenz unter Windows ausgeführt.

Programmablauf	Laufzeitmittelwert [s]	Laufzeitmittelwert [%]
sequenziell	83,7	100
parallel	20,5	25

Für die durchgeführte Parallelisierung konnte ein Beschleunigungsfaktor von 4 erreicht werden. Dieser Wert fällt deutlich geringer aus als der in Gl. (6.1) berechnete theoretische Beschleunigungsfaktor von 11,84. Es wird angenommen, dass der Hauptgrund dafür bei der erforderlichen Synchronisierung des Programmablaufs von TALdia liegt. Diese ist in der theoretischen Beschleunigung nicht berücksichtigt. Ein weiterer Grund wird in der nicht für parallele Threads optimierten Datenhaltung vermutet.

6.5 Prognostizierter Anpassungsbedarf für das Beispielprogramm ARTM

Aus dem erarbeiteten Konzept sowie dessen Umsetzung konnten Themen identifiziert werden, die bei der nachträglichen Parallelisierung eines komplexen Programms, wie es auch bei ARTM der Fall ist, im besonderen Maße berücksichtigt werden sollten. Diese sind:

- Anpassung des gesamten Ablaufs eines Programms auf die parallele Abarbeitung: Insbesondere führt die Einhaltung des EVA-Prinzips zu einem eher parallelisierbaren Programm.
- Datenstrukturen sollten für konkurrierende Operationen ausgelegt werden.
- Die Speicherung von Daten sollte thread-orientiert organisiert werden.

Für das Beispielprogramm ARTM bzw. TALdia bedeutet das im speziellen, dass für eine Parallelisierung von ARTM folgende Bereiche potenziell geändert werden müssten:

- Der Programmablauf in TALdia sollte umstrukturiert werden, so dass dieser näher am EVA-Prinzip orientiert ist.
- Die Datenstruktur der doppelt verketteten Liste (Table Manager) sollte für konkurrierende Operationen ertüchtigt werden.

- Datenstrukturen sollten, unter Berücksichtigung von „False Sharing“, in threadorientierten Datenstrukturen überführt werden.

Daneben existieren in Programmen, analog zu ARTM, noch weitere Aspekte, denen bei einer Parallelisierung Aufmerksamkeit geschenkt werden sollte. Dies sind beispielsweise:

- Die Anzahl der globalen und statischen Variablen sollte minimiert werden.
- Die Fehlerbehandlung muss für den parallelen Betrieb umgearbeitet werden.
- Das Programmlogbuch (Logging) muss für den parallelen Betrieb umgearbeitet werden.
- Die Fortschrittsanzeige muss für den parallelen Betrieb angepasst werden.
- Die Anzahl der zu benutzenden Prozessorkerne sollte vom Benutzer festgelegt werden können. Daneben muss die Anzahl der verwendeten Prozessorkerne zum bearbeiteten Problem und der Parallelisierung passen. Beispielsweise ergibt sich kein Mehrwert daraus, für ein Aufgabe mit neun parallelen Teilaufgaben einen Computer mit 24 Kernen vorzusehen.
- Die Generierung der Zufallszahlen sollte überdacht werden. Das Modul „MicroArchitecture Exploration“ vom Intel VTune Profiler hat gezeigt, dass die `do{…} while`-Schleife sowie die teuren mathematischen Funktonen `sqrt` und `log` das Programm verlangsamen. Aufgrund der Komplexität der Mikroarchitektur der CPU wurde dieser Bereich der Parallelisierung noch nicht optimiert. Ein erster Schritt wäre beispielsweise der Versuch, eine große Anzahl von Zufallszahlen mittels SIMD-Vektorisierung zu realisieren.

6.6 Zusammenfassung

Sowohl die Umsetzung der Parallelisierung in ARTM auf der Partikelebene als auch die Parallelisierung von TALdia führten zu einer Beschleunigung der jeweiligen Programme. Beide erreichten dabei unabhängig voneinander ein Beschleunigungsfaktor von drei bis vier bei einer Verwendung von zwölf Kernen.

Bei der Parallelisierung auf der Partikelebene spielten dabei insbesondere Prozesse im Millisekunden-Bereich eine große Rolle, die sich durch das Vermeiden von False-Sharing Effekten sowie angepassten Datenstrukturierungen optimieren ließ. Ebenso wurde

detailliert analysiert, inwieweit der Speicherzugriff erfolgt. Hierbei zeigte sich durch die Analyse mittels des Intel VTune Profilers und des integrierten Memory-Access-Tools, dass das Programm den Großteil des allokierten Speichers in den Cache lädt, welcher sich auf den schnellsten L1-Cache und dem langsameren L3-Cache primär verteilte. Der L2-Cache hatte einen sehr geringen Anteil. L1-Cache ist dabei relevant für die einzelnen Threads, während der L3-Cache als gemeinsamer Speicher für alle Threads dient, da er eine größere, aber langsamere Speicherebene ist als L1 und L2. Aufgrund der Nutzung des Cache anstelle des Arbeitsspeichers (RAM) wurde die Partikelschleife somit hinsichtlich der Nutzung des Speichers effizient optimiert. Neben dem Speicherzugriff war ebenfalls die Verwaltung und Zuordnung der Teilaufgaben der Threads sehr wichtig. Hier konnte das sogenannte Scheduling der Threads genutzt werden, d. h. das Vorgehen, wie die Iterationen in einer parallelen Umgebung aufgeteilt werden. Nach detaillierter Untersuchung wurde das Scheduling `guided` als zielführend ausgewählt.

Bei der Parallelisierung der Berechnung der Windfelder in TALdia war das Hauptproblem durch die Parallelisierung einer Schleife gegeben, die sich sehr weit oben, also am Anfang, in der Aufrufhierarchie befindet. Das führte dazu, dass in sehr vielen nachfolgend verwendeten Funktionen Änderungen nötig wurden. Hierbei musste besonders darauf geachtet werden, wie die Vielzahl von globalen Variablen verarbeitet wurde. Zusammen mit den Erkenntnissen aus der Parallelisierung des Transportmodells von ARTM konnten weitere potenzielle Optimierungsschritte identifiziert werden. Hierzu zählt die Anpassung der Datenspeicherung hin zu geeigneten threadspezifischen Strukturen, die auch das Prinzip des False-Sharing berücksichtigen. Darüber hinaus kann die Effizienz durch eine Optimierung des Programmablaufs gesteigert werden, wobei dieser sich stärker als bisher am EVA-Prinzip orientieren sollte, um Datenzugriffe auf der Festplatte, dem langsamsten verfügbaren lokalen elektronischen Speicher, zu minimieren.

7 Zusammenfassung

In diesem Eigenforschungsvorhaben wurde die Kompetenz der GRS im Bereich der Parallelisierung von Simulations- und Modellcodes erweitert. Als komplexes Simulationsprogramm wurde dafür exemplarisch das Atmosphärische Radionuklid Transport Modell (ARTM) herangezogen. Der Open-Source Charakter von ARTM ermöglicht einen leichten Zugang zum Quellcode. Darüber hinaus konnte, durch die bestehende Kenntnis des Programmablaufs, das Augenmerk von Beginn an auf die Methoden und Umsetzungsstrategien der Codeparallelisierung gelegt werden.

Im Eigenforschungsvorhaben wurde deshalb der internationale Stand von Wissenschaft und Technik zur Parallelisierung von Lagrange-Partikelmodellen oder ähnlichen Ausbreitungsmodellen recherchiert und bewertet. Neben den bisherigen Entwicklungen zur Effizienzsteigerung von ARTM wurden die Ausbreitungsmodelle MPTRAC, FLEXPART, PALM, HYSPLIT und PMSS näher analysiert. Bei den meisten Ausbreitungsmodellen wurden die parallelen Programmiermodelle OpenMP und MPI eingesetzt. Bei einigen fand auch eine unterstützende Beschleunigung durch GPUs statt. In jedem Ausbreitungsmodell betonten die Autoren die Komplexität, die das Vorhaben der Parallelisierung darstellte. Des Weiteren wurde eine Recherche zu hardwarenahen Parallelisierungsmöglichkeiten und dazu passenden Bibliotheken, Kompilierern und Grafikkarten (bei GPU-Parallelisierung) durchgeführt. Betrachtet wurden die parallelen Programmiermodelle OpenMP, MPI, OpenCL, OpenACC, CUDA, ROCm/HIP und Kokkos. Basierend auf diesen Erkenntnissen, den benötigten Kompilierern, den architektonischen Besonderheiten von Mehr-Kern-Prozessoren und GPUs und dem in ARTM vorliegenden Programmablauf und den Programmstruktureigenschaften wurde vom geplanten Vorhaben, die Parallelisierung durch GPU Beschleunigung zu erzielen, abgewichen, da ARTM nicht über ausreichende datenparallele Teilaufgaben verfügt. Diese sind allerdings Voraussetzung für eine GPU-Parallelisierung. Stattdessen wurde eine CPU-Parallelisierung angestrebt. Dazu wurden mögliche Stellen innerhalb des Quellcodes von ARTM, die für eine task-parallele Optimierung geeignet sind, identifiziert. Aufbauend darauf wurde ein Konzept erarbeitet, an welchen Stellen ARTM sinnvoll parallelisiert werden kann. Nach Umsetzung des Konzepts konnte für das Partikelmodel von ARTM und für TALdia jeweils ein Beschleunigungsfaktor um den Faktor 3 – 4 beobachtet werden. Aufgetretene Probleme wurden diskutiert. Die Simulationsergebnisse des sequenziellen Programms stimmten mit denen des Parallelen überein. Ausgehend von den Erkenntnissen aus diesem Eigenforschungsvorhaben konnte der generelle Anpassungsbedarf an komplexe Simulationsprogramme aber auch der spezielle Anpassungsbedarf an ARTM identifiziert

werden. Das Hauptaugenmerk ist demnach auf eine thread-orientierte Datenstrukturierung und die Einhaltung des EVA-Prinzips zu legen.

Die Bearbeitung dieses Eigenforschungsvorhabens konnte die Kompetenz der GRS im Bereich der Parallelisierung von komplexen Computerprogrammen deutlich erweitern. Es wird erwartet, dass die gewonnenen Erkenntnisse zukünftig zielführend bei der Entwicklung, Weiterentwicklung und Optimierung von Programmen eingesetzt werden können.

Literaturverzeichnis

/ADV 24a/ Advanced Micro Devices, Inc.: AMD ROCm documentation, ROCm™ Software 6.2.4. Stand vom 15. November 2024, erreichbar unter https://rocm.docs.amd.com/en/docs-6.2.4/, abgerufen am 29. November 2024.

/ADV 24b/ Advanced Micro Devices, Inc.: HIP Documentation, HIP 6.2.41134. Erreichbar unter https://rocm.docs.amd.com/projects/HIP/en/latest/, abgerufen am 29. November 2024.

/BAK 24/ Bakels, L., Tatsii, D., Tipka, A., Thompson, R., Dütsch, M., Blaschek, M., Seibert, P., Baier, K., Bucci, S., Cassiani, M., Eckhardt, S., Groot Zwaaftink, C., Henne, S., Kaufmann, P., et al.: FLEXPART version 11: improved accuracy, efficiency, and flexibility. Geoscientific Model Development, Bd. 17, Nr. 21, S. 7595–7627, DOI 10.5194/gmd-17-7595-2024, 2024.

/BAR 17/ Barney, B., Frederick, D.: Introduction to Parallel Computing Tutorial. Lawrence Livermore National Laboratory, erreichbar unter https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial, abgerufen am 15. November 2024.

/BAR 23/ Barlas, G.: Chapter 6 - GPU programming: CUDA. In: Barlas, G. (Hrsg.): Multicore and GPU Programming (Second Edition). S. 389–581, ISBN 978-0-12-814120-5, DOI 10.1016/B978-0-12-814120-5.00015-9, Morgan Kaufmann: Philadelphia, 2023.

/BAR 24/ Barney, B.: Message Passing Interface (MPI). Lawrence Livermore National Laboratory, erreichbar unter https://hpc-tutorials.llnl.gov/mpi/, abgerufen am 2. Januar 2025.

/COO 12/ Cook, S.: CUDA Programming : A Developer's Guide to Parallel Computing with GPUs. ISBN 9780124159884, Elsevier Science & Technology: Chantilly, UNITED STATES, 2012.

/GCC 24/ GCC Wiki: OpenACC. Stand vom 28. Juni 2024, erreichbar unter https://gcc.gnu.org/wiki/OpenACC, abgerufen am 26.11.2024.

/GRE 02/ Greenwald, M.: Two-handed emulation. In: ACM: Proceedings of the twenty-first annual symposium on Principles of distributed computing. New York, NY, USA, S. 260–269, DOI 10.1145/571825.571874: New York, NY, USA, 2002.

/GRS 20/ Richter, C., Thielen, H., Spieker, K.: Weiterentwicklung des atmosphärischen Ausbreitungsmodells ARTM bezüglich weiterer Anwendungsbereiche, Windfeld- und Grenzschichtmodell, Dokumentation, Abschlussbericht zum Vorhaben 3616S72575. Hrsg.: Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH, GRS-A-Bericht, GRS-A-3973, 153 S.: Köln, Oktober 2020.

/HAK 04/ Hakan Sundell: Efficient and practical non-blocking data structures. Dissertation, Department of Computing Science, Chalmers University of Technology and Göteborg University: Göteborg, 2004.

/HAR 01/ Harris, T. L.: A Pragmatic Implementation of Non-blocking Linked-lists. In: Welch, J. (Hrsg.): Distributed Computing, 15th International Conference, DISC 2001 Lisbon, Portugal, October 3-5, 2001 Proceedings. International Symposium on Distributed Computing, Lecture Notes in Computer Science, Nr. 2180, S. 300–314, ISBN 978-3-540-45414-4, DOI 10.1007/3-540-45414-4_21, Springer: Berlin, Heidelberg, 2001.

/HAR 17/ Harris, M.: An Even Easier Introduction to CUDA. NVIDIA Corporation, Stand vom 25. Januar 2017, erreichbar unter https://developer.nvidia.com/blog/even-easier-introduction-cuda/, abgerufen am 29. November 2024.

/HER 23/ Herten, A.: Many Cores, Many Models: GPU Programming Model vs. Vendor Compatibility Overview. In: Association for Computing Machinery: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W 2023: Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, Denver CO USA, 12. - 17. November 2023, ACM Digital Library, S. 1019–1026, ISBN 9798400707858, DOI 10.1145/3624062.3624178: Erscheinungsort nicht ermittelbar, 2023.

/HOF 16/ Hoffmann, L., Rößler, T., Griessbach, S., Heng, Y., Stein, O.: Lagrangian transport simulations of volcanic sulfur dioxide emissions: Impact of meteorological data products. Journal of Geophysical Research: Atmospheres, Bd. 121, Nr. 9, S. 4651–4673, DOI 10.1002/2015JD023749, 2016.

/HOF 22/ Hoffmann, L., Baumeister, P. F., Cai, Z., Clemens, J., Griessbach, S., Günther, G., Heng, Y., Liu, M., Haghighi Mood, K., Stein, O., Thomas, N., Vogel, B., Wu, X., Zou, L.: Massive-Parallel Trajectory Calculations version 2.2 (MPTRAC-2.2): Lagrangian transport simulations on graphics processing units (GPUs). Geoscientific Model Development, Bd. 15, Nr. 7, S. 2731–2762, DOI 10.5194/gmd-15-2731-2022, 2022.

/HOF 24/ Hoffmann, L., Haghighi Mood, K., Herten, A., Hrywniak, M., Kraus, J., Clemens, J., Liu, M.: Accelerating Lagrangian transport simulations on graphics processing units: performance optimizations of Massive-Parallel Trajectory Calculations (MPTRAC) v2.6. Geoscientific Model Development, Bd. 17, Nr. 9, S. 4077–4094, DOI 10.5194/gmd-17-4077-2024, 2024.

/IBM 24/ IBM: What is accelerated computing? erreichbar unter https://www.ibm.com/think/topics/accelerated-computing, abgerufen am 14. November 2024.

/INT 24/ Intel Corporation: On the Migration of OpenACC* API to OpenMP* API. Erreichbar unter https://www.intel.com/content/www/us/en/developer/articles/technical/migration-of-openacc-api-to-openmp-api.html, abgerufen am 26. November 2024.

/INT 25a/ Intel Corporation: Intel® VTune™ Profiler Documentation. Erreichbar unter https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler-documentation.html?utm_source=chatgpt.com, abgerufen am 4. September 2025.

/INT 25b/ Intel Corporation: FPGA im Vergleich zu GPU bei Deep-LearningAndensungen. Erreichbar unter https://www.intel.de/content/www/de/de/fpga-solutions/artificial-intelligence/fpga-gpu.html, abgerufen am 8. April 2025.

/INT 25c/ Intel Corporation: Intel® VTune™ Profiler 2025.4.0. Stand vom 18. August 2025, erreichbar unter https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.i6xhgk, abgerufen am 1. September 2025.

/INT 25d/ Intel Corporation: How Intel® Core™ Processors Work. Erreichbar unter https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html, abgerufen am 04.09.02025.

/KAE 15/ Kaeli, D. R., Mistry, P., Schaa, D., Zhang, D. P.: Heterogeneous Computing with OpenCL 2. 0. ISBN 9780128016497, Elsevier Science & Technology: Chantilly, UNITED STATES, 2015.

/KAN 22/ Kanetkar, Y.: Data Structures Through C. 1. Aufl., 1294 S., ISBN 9789355511904, BPB Publications: Los Angeles, 2022.

/KHR 24/ Khronos Group (Hrsg.): OpenCL. Beaverton, erreichbar unter https://www.khronos.org/opencl/, abgerufen am 28. November 2024.

/KNO 16/ Knoop, H., Gronemeier, T., Knigge, C., Steinbach, P.: Porting the MPI Parallelized LES Model PALM to Multi-GPU Systems - An Experience Report. In: Taufer, M., Mohr, B., Kunkel, J. M. (Hrsg.): High Performance Computing. S. 508–523, ISBN 978-3-319-46079-6, Springer International Publishing: Cham, 2016.

/KOK 19/ Kokkos Project: KOKKOS ECOSYSTEM, A Linux Foundation Project. Erreichbar unter https://kokkos.org/, abgerufen am 14. November 2024.

/KRE 24/ Krekel H., P.-D. T.: Pytest, pytest: helps you write better programs. Stand vom Januar 2024, erreichbar unter https://docs.pytest.org/, abgerufen am 29. Januar 2024.

/MAR 15/ Maronga, B., Gryschka, M., Heinze, R., Hoffmann, F., Kanani-Sühring, F., Keck, M., Ketelsen, K., Letzel, M. O., Sühring, M., Raasch, S.: The Parallelized Large-Eddy Simulation Model (PALM) version 4.0 for atmospheric and oceanic flows: model formulation, recent developments, and future perspectives. Geoscientific Model Development, Bd. 8, Nr. 8, S. 2515–2551, DOI 10.5194/gmd-8-2515-2015, 2015.

/MIC 25/ Microsoft: Übersicht über die Profilerstellungstools (C#, Visual Basic, C++, F#). Stand vom 12. Juni 2025, erreichbar unter https://learn.microsoft.com/de-de/visualstudio/profiling/profiling-feature-tour?view=vs-2022&pivots=programming-language-dotnet, abgerufen am 21. Juli 2025.

/MOR 23/ Morton, D.: Performance Enhancement of Flexpart Atmospheric Transport Model for GPU Environment. Präsentation, CTBT: Science and Technology Conference 2023, CTBTO: Wien, 19. - 23. Juni 2023.

/NAT 24/ National Technology & Engineering Solutions of Sandia: 4. Compiling. Erreichbar unter https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/Compiling.html, abgerufen am 14. November 2024.

/NAV 14/ Navarro, C. A., Hitschfeld-Kahler, N., Mateu, L.: A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures. Communications in Computational Physics, Bd. 15, Nr. 2, S. 285–329, DOI 10.4208/cicp.110113.010813a, 2014.

/NEW 23/ New Mexico State University: Message Passing Interface. Erreichbar unter https://hpc.nmsu.edu/discovery/mpi/introduction/, abgerufen am 2. Januar 2025.

/NVI 24a/ NVIDIA: NVIDIA Glossary. Stand von 2024, erreichbar unter https://www.nvidia.com/en-us/glossary/, abgerufen am 27. November 2024.

/NVI 24b/ NVIDIA: HPC SDK Documentation, NVIDIA HPC Compilers User's Guide. Stand vom 13. November 2024, erreichbar unter https://docs.nvidia.com/hpc-sdk//compilers/hpc-compilers-user-guide/index.html, abgerufen am 15. November 2024.

/NVI 24c/ NVIDIA Corporation: CUDA Release Notes, Release 12.6. 14. November 2024.

/OH 12/ Oh, F.: What Is CUDA? Hrsg.: NVIDIA Corporation, Stand vom 10. September 2012, erreichbar unter https://blogs.nvidia.com/blog/what-is-cuda-2/ , abgerufen am 29. November 2024.

/OLD 11/ Oldrini, O., Olry, C., Moussafir, J., Armand, P., Duchenne, C.: Development of PMSS, the parallel version of MICRO-SWIFT-SPRAY. In: HARMO 14th Local Organizing Committee (Hrsg.): 14th Conference on Harmonisation within Atmospheric Dispersion Modelling for Regulatory Purposes. Kos, Griechenland, 2011.

/OLD 19/ Oldrini, O., Armand, P., Duchenne, C., Perdriel, S.: Parallelization Performances of PMSS Flow and Dispersion Modeling System over a Huge Urban Area. Atmosphere, Bd. 10, Nr. 7, DOI 10.3390/atmos10070404, 2019.

/OPE 19/ OpenMP: OpenMP, The OpenMP API specification for parallel programming. Erreichbar unter https://www.openmp.org/, abgerufen am 14. November 2024.

/OPE 23b/ OpenCL: OpenCL Guide. Hrsg.: Khronos Group, Khonos Group, Stand vom 3. März 2023, erreichbar unter https://github.com/KhronosGroup/OpenCL-Guide/tree/main, abgerufen am 28. November 2024.

/OPE 23a/ OpenACC.org: OpenACC Programming and Best Practices Guide. Stand von 2023, erreichbar unter https://openacc-best-practices-guide.readthedocs.io/en/latest/index.html, abgerufen am 26. November 2024.

/PIS 19/ Pisso, I., Sollum, E., Grythe, H., Kristiansen, N. I., Cassiani, M., Eckhardt, S., Arnold, D., Morton, D., Thompson, R. L., Groot Zwaaftink, C. D., Evangeliou, N., Sodemann, H., Haimberger, L., Henne, S., et al.: The Lagrangian particle dispersion model FLEXPART version 10.4. Geoscientific Model Development, Bd. 12, Nr. 12, S. 4955–4997, DOI 10.5194/gmd-12-4955-2019, 2019.

/PLL 17/ Pllana, S., Xhafa, F.: Programming Multicore and Many-Core Computing Systems. ISBN 9781119331995, John Wiley & Sons, Incorporated: Newark, UNITED STATES, 2017.

/RAA 01/ Raasch, S., Schröter, M.: PALM - A large-eddy simulation model performing on massively parallel computers. Meteorologische Zeitschrift, Bd. 10, Nr. 5, S. 363–372, DOI 10.1127/0941-2948/2001/0010-0363, 2001.

/RAU 07/ Rauber, T., Rünger, G.: Parallele Programmierung. EXamen.press, 2. Aufl., 485 S., ISBN 978-3-540-46549-2, Springer: Berlin, Heidelberg, 2007.

/SHE 13/ Sheikh, N. (Hrsg.): Implementing Analytics : MK Series on Business Intelligence. ISBN 978-0-12-401696-5, Morgan Kaufmann: Boston, 2013.

/SNY 21/ Snyder, J.: The Kokkos EcoSystem. In: Dennig, Y. (Hrsg.), S. 34–36, Sandia National Laboratories, März 2021: A new decade of high performance computing 2020.

/TEX 18/ Texas Insturments: TI OpenCL User's Guide 1.2.0, Compilation. Hrsg.: Texas Instruments Incorporated: Texas, Stand vom 30. Dezember 2019, erreichbar unter https://downloads.ti.com/mctools/esd/docs/opencl/compilation.html, abgerufen am 28. November 2024.

/TOP 24/ TOP500.org: TOP500, JUNE 2014. Stand vom Juli 2024, erreichbar unter https://www.top500.org/lists/top500/2024/06/, abgerufen am 18. November 2024.

/TRO 18/ Trobec, R., Slivnik, B., Bulić, P., Robič, B.: Introduction to parallel computing, From algorithms to programming on state-of-the-art platforms. Undergraduate topics in computer science, 256 S., ISBN 9783319988320, Springer: Cham, 2018.

/UNI 20/ University of Hamburg (Hrsg.): Parallelization Overheads. Stand vom 22. Juni 2020, erreichbar unter https://www.hhcc.uni-hamburg.de/learning-hpc/getting-started-with-hpc-clusters-b/getting-started-with-hpc-clusters-b-y-parallelization-overheads-b.html, abgerufen am 26. August 2025.

/VAL 95/ Valois, J. D.: Lock-free linked lists using compare-and-swap. In: ACM Press: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing - PODC '95. New York, New York, USA, S. 214–222, DOI 10.1145/224964.224988: New York, New York, USA, 1995.

/WOL 25/ Wolfman, S.: What is gprof? Duke University, erreichbar unter https://users.cs.duke.edu/~ola/courses/programming/gprof.html, abgerufen am 21. Juli 2025.

/WOL 19/ Wolf, J.: C von A bis Z, Das umfassende Handbuch. Rheinwerk Computing, 3. Aufl., 1190 S., ISBN 9783836214117, Rheinwerk Verlag: Bonn, 2019.

/YU 19/ Yu, F., Strazdins, P., Henrichs, J., Pugh, T.: Shared Memory and GPU Parallelization of an Operational Atmospheric Transport and Dispersion Application. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). S. 729–738, DOI 10.1109/IPDPSW.2019.00121, 2019.

/ZEN 24/ Zenodo (Hrsg.): Massive-Parallel Trajectory Calculations (MPTRAC). DOI 10.5281/zenodo.12751121, Forschungszentrum Jülich, Stand vom 12. Juli 2024, erreichbar unter https://zenodo.org/records/12751121, abgerufen am 25. Oktober 2024.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ALU	Arithmetic Logic Unit
AoSs	Array of Structures
ARTM	Atmosphärisches Radionuklid Transport Modell
CC-NUMA	Cache Coherent-NUMA
CC-UMA	Cache Coherent-UMA
CPU	Central Processing Unit
CTBO	Comprehensive Nuclear-Test-Ban Treaty Organization
CTBT	The Comprehensive Nuclear-Test-Ban Treaty
DDM	Domain Decomposition Methode
DSP	Digital Signal Processor
FLEXPART	FLEXible PARTicle Dispersion Model
FPGA	field programmable gate array
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphic Processor Unit
HIP	Heterogeneous-computing Interface for Portability
HPC	High Performance Computer
HYSPLIT	Hybrid Single-Particle Lagrangeian Integrated Trajectory Model
IAS	Institute for Advanced Simulation
JSC	Jülich Supercomputing Centre
LES	Large-Eddy Simulationen
MIMD	Multiple Instructions Multiple Data
MISD	Multiple Instructions Single Data
MPI	Message Passing Interface
MPTRAC	Massive-Parallel Trajectory Calculations
MTP	Multiple-Tiles Parallelization
NUMA	Non-Uniform Memory Access
Open-MP	Open Multi Processing

PALM	PArallelized Large-Eddy Simulation Model
PCI-E	Peripheral Component Interconnect Express
PE	Prozessoreinheiten
PMSS	Parallel Micro-SWIFT-SPRAY
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SM	streaming multiprocessor
SMP	Symmetric Multiprocessor
SMPIC	Specific MPI communicators
SMT	Simultanes Multithreading
SoAS	Structure of Arrays
SP	Stream Processor
STP	Single-Tile Parallelization
TCP	Transmission Control Protocol
TP	time frame parallelization
UMA	Uniform Memory Access

Gesellschaft für Anlagen- und Reaktorsicherheit (GRS) gGmbH

Schwertnergasse 1
50667 Köln
Telefon +49 221 2068-0
Telefax +49 221 2068-888

Boltzmannstraße 14
85748 Garching b. München
Telefon +49 89 32004-0
Telefax +49 89 32004-300

Kurfürstendamm 200
10719 Berlin
Telefon +49 30 88589-0
Telefax +49 30 88589-111

Theodor-Heuss-Straße 4
38122 Braunschweig
Telefon +49 531 8012-0
Telefax +49 531 8012-200

www.grs.de

ISBN 978-3-911727-13-6