Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Parallelisierungsansätze für Wissenschaftliches Rechnen anhand numerischer Simulation

vorgelegt von

Rasmus Pranke

MIN-Fakultät

Fachbereich Informatik

Studiengang: Informatik

Matrikelnummer: 6920830

Abgabedatum: 26. September 2022

Erstgutachter: Jannek Squar

Zweitgutachter: Prof. Dr. Thomas Ludwig

Betreuer: Jannek Squar, Anna Fuchs

Abstract

Computersimulation ist ein wichtiger Bestandteil moderner Wissenschaft. Es ist eine der größten Herausforderungen für wissenschaftliche Programmierer, diesen eine hohe Performanz zu verliehen. Parallelisierung ist ein wichtiges Werkzeug, um diese Performanz zu erreichen.

In dieser Arbeit wird eine repräsentatives Programm erarbeitet, dass als Grundlage für den Einstieg in dieses Feld dienen kann. Das N-Körper-Problem als geeignetes Beispiel identifiziert. Anhand diesem werden die relevanten Ansätze paralleler Ausführung und Ausgabe illustriert. Dabei werden POSIX-Threads, OpenMP und das Message-Passing-Interface als praxisrelevante Werkzeuge für die parallele Ausführung und NetCDF sowie HDF5 für parallele Ein- und Ausgabe vorgestellt.

Eine Lösung für das N-Körper-Problem wird implementiert und mithilfe dieser Ansätze parallelisiert. Die Ansätze werden gegenübergestellt und im Hinblick auf ihre Komplexität und Performanz verglichen. Anhand des implementierten Programms werden typische Performanz-Eigenschaften paralleler Software aufgezeigt. Abschließend wird eine Reihe an möglichen Übungsaufgaben identifiziert, mit denen das Programm in der Lehre verwendet werden kann.

Es ergibt sich, dass alle Ansätze eine enorme Performanz bieten, dabei aber unterschiedliche Schwerpunkte setzen. OpenMP erlaubt es, die gleiche Parallelisierung wie POSIX-Threads auf Kosten von Flexibilität zu erreichen. MPI dagegen setzt ein etwas komplexere Modell um, dass dafür sehr einfach die Skalierung über einen Rechner hinaus erlaubt. NetCDF und HDF5 stehen sich als Abwägung von Möglichkeiten und Komplexität gegenüber, in der NetCDF durch Simplizität und HDF5 durch Macht glänzt.

Inhaltsverzeichnis

1. Einführung

Die Berechnung komplexer Modelle mit riesigen Datensätzen stellt einen zentralen Aspekt moderner Wissenschaft dar. Simulation wurde sogar als „dritte Säule der Wissenschaft“ bezeichnet [32]. Umso wichtiger ist es, diese Berechnungen zu beschleunigen. Mit diesem Problem beschäftigt sich das Feld des Hochleistungsrechnens. Mehr und mehr wird dafür Parallelisierung als eines der wichtigsten Werkzeuge eingesetzt. Moderne Supercomputer sind darauf optimiert, möglichst viele Operationen parallel zueinander auszuführen. Entsprechende Software-Bibliotheken und Schnittstellen werden laufend weiterentwickelt und ausgereizt. Posix-Threads [24], OpenMP [21] und MPI [16] stellen dabei wichtige Ansätze parallelen Rechnens dar.

Die Datenmengen, die aus diesen Modellen generiert werden, stellen das Feld vor ein Performanz-Problem. Häufig ist die Datenrate des Ein- und Ausgabesystems ein Flaschenhals für die Geschwindigkeit der Berechnungen. Verschiedene Software-Bibliotheken stellen Möglichkeiten bereit, Ausgabe zu parallelisieren und ihre Performanz an die der Berechnungen anzupassen. Dazu zählen insbesondere die HDF5-Bibliothek [30] und die NetCDF-Bibliothek [19].

Um den Zugang zu einem solchen komplexen und tiefgreifenden Feld zu ermöglichen, ist es wichtig, die grundlegenden praktischen Ansätze und die für sie relevanten theoretischen Grundlagen zu veranschaulichen. Dieses Ziel setzt sich diese Arbeit. Zu diesem Zweck wird ein Problem identifiziert, anhand dessen die wichtigsten Ansätze implementiert und die theoretischen Grundlagen aufgezeigt werden können. Ein dieses Problem lösendes Programm wird implementiert. Die Umsetzung wird quantitativ wie qualitativ analysiert. Der Effekt und die Umsetzung der Ansätze werden miteinander verglichen und Stärken und Schwächen identifiziert. Das Programm wird als Lehrmaterial analysiert und es werden Aufgaben für Studierende formuliert, um ihnen anhand des Programms den Zugang zu dem Feld der Parallelisierung zu ermöglichen.

Diese Arbeit orientiert sich dabei an Werken, die Parallelisierung aufschlüsseln und sich auch im Konkreten mit der Umsetzung befassen. Dazu gehört die Arbeit der *Partnership for Advanced Computing in Europe* [22], kurz PRACE, deren Reihe an Best Pratice Guides jeweils einen Teilbereich des Hochleistungsrechnens aufarbeiten und praktisch zugänglich machen. Die Arbeit *Paralllel I/O* [15] dient als Grundlage für die Umsetzung der Datenausgabe. Vor kurzem wurde außerdem das Buch *Principles of Parallel Scientific Computing* [32] veröffentlicht, welches mit dem Anspruch der Lehre die theoretische Seite der Parallelisierung anhand des N-Körper-Problems beleuchtet.

2. Hintergrund

In diesem Kapitel werden die grundlegenden Konzepte erläutert und die verwendeten Softwarebibliotheken vorgestellt. Erst wird die theoretische Grundlage für paralleles Rechnen behandelt, dann der Aufbau moderner Supercomputer und schließlich die auf diesen verwendete Software.

2.1. Paralleles Rechnen

2.1.1. Speedup und das Amdahlsche Gesetz

Speedup bezeichnet den Zusammenhang zwischen der sequentiellen und der parallelen Ausführungszeit eines Programms. Er gibt an, das wievielfache der Geschwindigkeit in der parallelen Ausführung erzielt wird. Speedup wird durch t_1/t_p definiert, wobei t_1 die sequentielle und t_p die parallele Ausführungszeit beschreibt. Wenn ein Programm sequentiell eine Ausführunsgzeit von 20 Sekunden hat, und bei einer Parallelisierung mit vier Prozessoren eine Ausführungszeit von 5 Sekunden erreicht, dann ist der Speedup $\frac{20}{5} = 4$.

Das Amdahlsche Gesetz beschreibt den maximalen Speedup, der von einer parallelen Ausführung zu erwarten ist. Sei t_s der Anteil der Ausführungszeit eines sequentiellen Programms, der nur sequentiell ausführbar ist. Dann ist $t_p = 1 - t_s$ der Anteil, der parallelisierbar ist. Unter der Annahme, dass die Parallelisierung keine zusätzlichen Kosten verursacht, ergibt sich für den höchstmöglichen Speedup das Amdahlsche Gesetz:

$$s = \frac{1}{1 - t_p} \tag{2.1}$$

Das heißt, wenn beispielsweise $t_p = 95\%$ eines Programms theoretisch parallelisierbar sind, dann ist der bestmögliche Speedup 20. Sei die Ausführungszeit $t = 20h$, dann ist $t_s = 0.95 * 20h = 1h$. Wenn durch Parallelisierung mit unendlich vielen Prozessoren die verbleibenden 19 Stunden sofort abgearbeitet werden, so verbleibt immer mindestens eine Stunde sequentielle Ausführung. Damit kann Speedup in diesem Fall 20 und niemals mehr betragen.

Abbildung 2.1 zeigt Speedup-Kurven für verschiedene Werte für t_p. Es ist deutlich zu sehen, wie der potentielle Speedup mit steigendem t_p zunimmt, und gleichzeitig der Gewinn pro Prozessor mit steigender Prozessorzahl abnimmt.

Wenn ein Programm vollständig parallelisierbar ist, dann ist der maximale Speedup gleich der Anzahl der eingesetzten Prozessoren. Nach dem Amdahlschen Gesetzt ergibt sich:

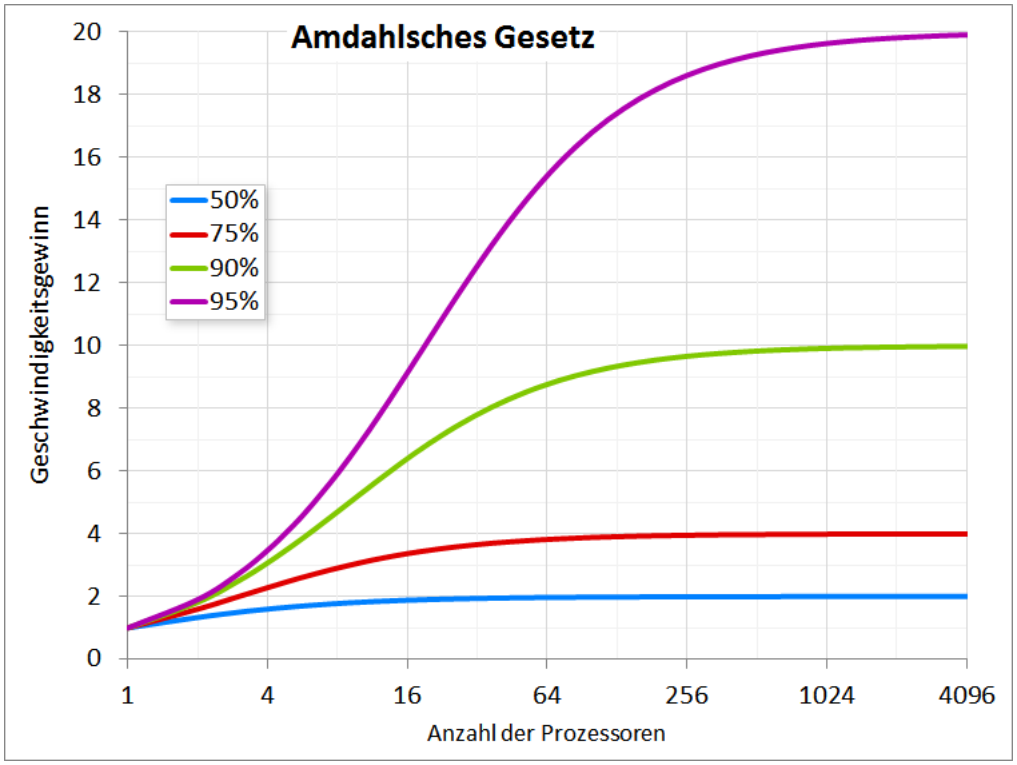

Abbildung 2.1.: Illustration des Amdahlschen Gesetzes [13]

$$\lim_{(t_p)\to 1} \left(\frac{1}{1-t_p}\right) = \infty \tag{2.2}$$

D.h. mit unendlich Prozessoren kann ein unendlicher Speedup erreicht werden.

Indem für ein gegebenes Problem der parallelisierbare Anteil geschätzt wird, kann die Effektivität der tatsächlich vorgenommenen Parallelisierung eingeschätzt werden. Unter der Annahme, das der Speedup des parallelisierbaren Anteils mit jedem weiteren Prozessor linear verbessert wird, ergibt sich ein charakteristisches Beschleunigungsverhalten, das in 2.1 illustriert ist.

2.1.2. E/A-gebunden vs Prozessor-gebunden

Es kann passieren, dass ein Programm Daten lädt oder speichert und die Ausführung auf das Abschließen dieser Operationen warten muss, bevor sie fortgesetzt werden kann. Wenn die Ausführungszeit eines Programms maßgeblich von einem solchen Umstand beeinflusst wird, spricht man von einem E/A-gebundenen Prozess.

Andersherum wird ein Programm Prozessor-gebunden genannt, wenn die Ausführungszeit stattdessen eher von der Geschwindigkeit des Prozessors bestimmt wird.

Da die Verbesserung von E/A-Performanz und Prozessor-Performanz oft unabhängig voneinander ist, lässt sich anhand dieser Eigenschaft feststellen, in welchem der beiden Bereiche Performanzverbesserungen einen größeren Effekt auf die Gesamtlaufzeit erzielen können. Insbesondere kann parallele Ausführung von E/A und Berechnungen dazu führen, dass ein Performanzgewinn in dem Bereich, in dem das Programm nicht gebunden ist, gar keinen Effekt auf die Gesamtlaufzeit hat.

2.1.3. Taxonomie nach Flynn

Grundsätzlich können vier Arten der parallelen Verarbeitung unterschieden werden, anhand dessen, ob sie sich auf die Daten oder die Anweisungen beziehen. Diese Unterscheidung wurde 1966 von Michael J. Flynn aufgestellt und ist bis heute relevant. Die Arten sind:

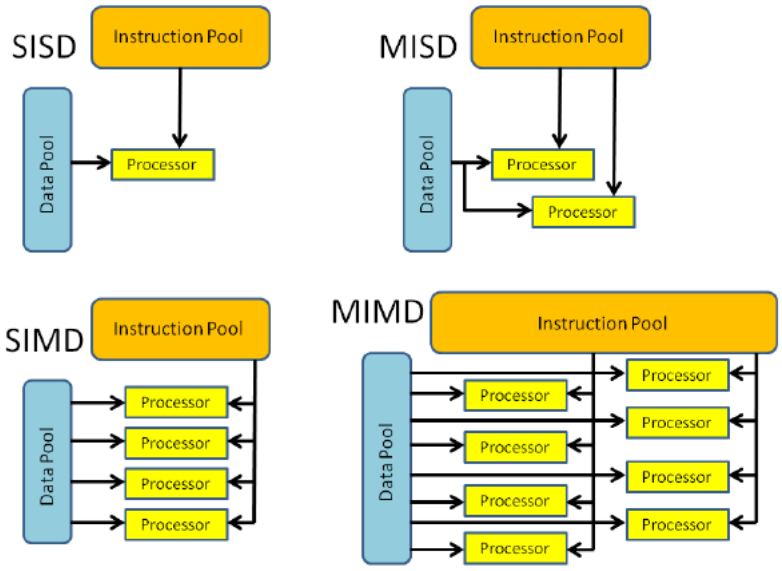

Abbildung 2.2.: Graphische Darstellung der Taxonomie nach Flynn. [11]

- SISD (Single Instruction, Single Data; Einzelne Anweisung, Einzelne Daten): Ein einzelner Anweisungsstrom wird auf einem einzelnen Datenstrom ausgeführt.
- SIMD (Single Instruction, Multiple Data; Einzelne Anweisung, Viele Daten): Der gleiche Anweisungsstrom wird parallel für viele Datenströme ausgeführt.
- MISD (Multiple Instruction, Single Data; Viele Anweisungen, Einzelne Daten): Für einen Datenstrom werden mehrere Anweisungsströme ausgeführt.

- MIMD (Multiple Instruction, Multiple Data; Viele Anweisungen, Viele Daten): Verschiedene Anweisungsströme werden parallel auf verschiedenen Datenströmen ausgeführt.

In die SISD-Kategorie fallen Einkernprozessoren, welche sequentiell einen Befehlsstrom verarbeiten. Moderne Computer, insbesondere Supercomputer, fallen in die MIMD-Kategorie. Ihre Vielzahl an Prozessorkernen arbeiten unabhängig an verschiedenen Daten. Moderne Prozessoren weisen außerdem Vektoranweisungen auf, die in SIMD einzuordnen sind und der Performanz dienen. Es ist umstritten, ob es Systeme gibt, die in MISD einzuordnen sind.

2.1.4. Threads und Prozesse

In modernen Computern wird Parallelisierung über Prozesse und Threads erreicht. Der POSIX-Standard, der sowohl Linux- als auch Macintosh-Systemen zugrunde liegt, definiert *Prozess* als ausführende Instanz eines Programms, dessen Speicherbereich von dem anderer Prozesse getrennt ist. Ein Prozess besteht aus einem oder mehreren Threads, welche jeweils Zugriff auf den Speicherbereich des Prozesses haben, aber unterschiedliche Aufrufstapel (engl. *Stacks*) verwenden. Threads und Prozesse können, sofern die ausführende Hardware das unterstützt, parallel zueinander ausgeführt werden.

Im Rahmen von Threads ist ein wichtiges Modell das sogenannte Fork-Join-Modell. Hier ist die Idee, dass ein Prozess neue Threads startet und Arbeit verrichten lässt und anschließend mit einem sogenannten *Join* auf den Abschluss dieser Threads wartet, um ihre Ergebnisse zu nutzen. Abbildung 2.3 zeigt dieses Modell.

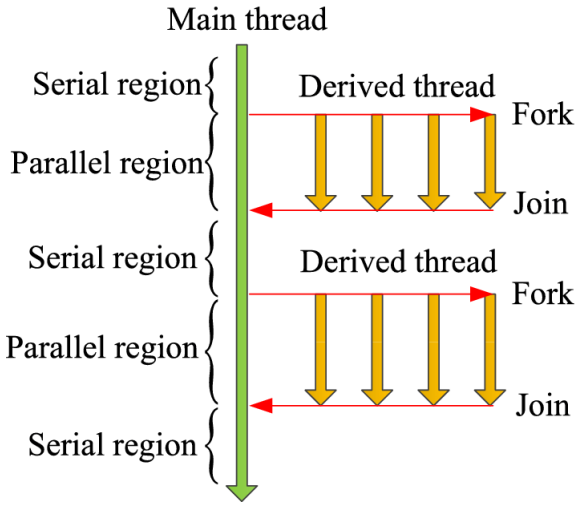

Abbildung 2.3.: Illustration des Fork-Join-Modells. [9]

2.1.5. Supercomputer

In modernen Supercomputern wird massive parallele Performanz durch die Zusammenarbeit hunderter Rechenknoten in einem sogenannten *Cluster* erreicht. Jeder Knoten verfügt über eigenen Arbeitsspeicher und kann unabhängig von allen anderen Knoten arbeiten. Über Hochgeschwindigkeitsschnittstellen können individuelle Knoten Daten austauschen und somit gemeinsam am gleichen Problem rechnen. Diese Architekturen erreichen heutzutage eine Performanz von bis zu knapp über einer Trillion Fließkommaoperationen pro Sekunde. [28]

Typischerweise werden diese Rechner über einen sogenannten Login-Knoten angesprochen, die selber nicht mitrechnet und für die Verwaltung von Software, Daten und Nutzern verwendet wird. Von ihr aus werden Vorbereitungen getroffen und über einen Scheduler Programme auf dem Cluster ausgeführt.

Die linke Seite der Abbildung 2.4 zeigt diesen Aufbau beispielhaft am Thor-Cluster.

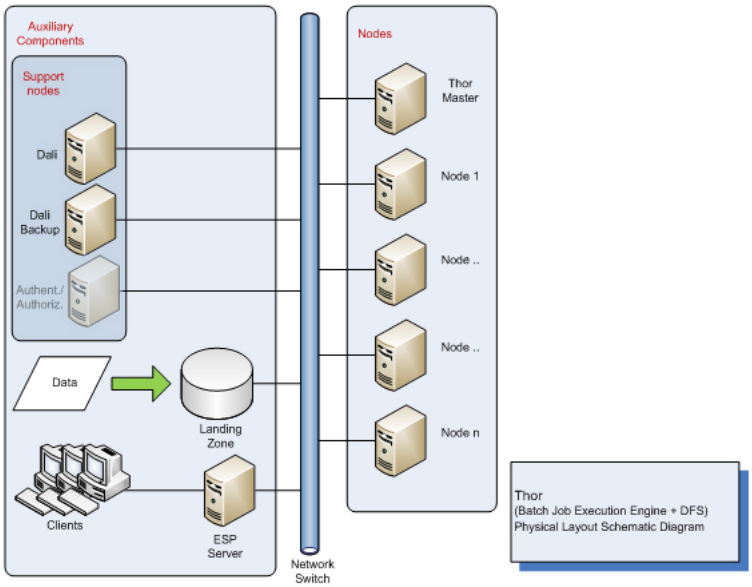

Abbildung 2.4.: Graphische Darstellung des Thor-Clusters als Beispiel für die Architektur von Supercomputern. [5] Der ESP-Server ist dabei der Login-Knoten.

Ein solcher Scheduler ist Slurm. Slurm ist ein vielseitig anwendbare Ressourcenverwaltungssoftware für zusammenhängende Cluster aus Linux-Computern. [34] Mit Slurm können Knoten verwaltet und angesprochen werden. Aus Anwendersicht dient Slurm vor allem als Werkzeug, um Programme auf mehreren Knoten eines Clusters gleichzeitig auszuführen. Dafür werden sogenannte *Batch-Skripte* verwendet. Sie bestehen aus einem Konfigurationsteil und einem auszuführenden Shell-Skript. Die Konfiguration weist Slurm an, welche und wie viele Knoten verwendet werden sollen, wie viele Tasks pro Knoten ausgeführt werden sollen, wie lange der Job laufen darf und vieles weiteres. Wenn

ein Batch-Skript in Auftrag gegeben wird, wird es asynchron einer Job-Warteschleife hinzugefügt. Sobald die für die Ausführung benötigten Ressourcen verfügbar sind, wird der Job auf die Knoten hochgeladen und ausgeführt. Administratorseitig erlaubt Slurm, Knoten zu aktivieren und zu deaktivieren, oder sie für eine bevorstehende Wartung zu leeren. Dann werden keine neuen Jobs entgegengenommen, bis alle Jobs verarbeitet wurden, und anschließend wird der Knoten deaktiviert.

2.1.6. Parallele Ein- und Ausgabe

Die enorme Performanz von Supercomputern wird häufig durch die Ausgabegeschwindgkeit ausgebremst. So erzeugen zum Beispiel Klimamodelle hunderte Terrabyte an Daten, die gespeichert werden müssen. Entsprechende Datenraten werden durch ein Zusammenspiel mehrerer Bestandteile, sowohl Software als auch Hardware, ermöglicht. Auf der Seite der Software finden sich parallele E/A-Bibliotheken, parallele Dateisysteme und Betriebssystemkomponenten. Hardwareseitig werden mehrere E/A-Knoten bereitgestellt, die direkt an das Cluster angeschlossen sind und in einem Speichernetzwerk aus Speicherknoten auf Speichergeräten Daten abspeichern. Dafür wird ein passendes Dateisystem benötigt, dass diese parallelen Zugriffe auf Seite der rechnenden Knoten koordiniert. Die beiden häufigsten sind dabei Lustre [14] und GPFS [10]. Auf den Speichergeräten selber wird ein ordinäres Dateisystem eingesetzt, dass von den parallelen Dateisystemen angesprochen wird. Es erfordert Software, die an die verwendeten Softwarekomponenten sowie die vorhandene Infrastruktur angepasst ist, um diese Ressourcen auszureizen.

Es werden vier Ansätze für parallele Ausgabe unterschieden.

- „Eine Datei, ein Schreiber“ (SFSW; „Single File Single Writer“)): Hierbei werden die zu schreibenden Daten in einem Prozess gesammelt und sequentiell geschrieben. In der sequentiellen Konfiguration ist es der einzige vorhandene Ansatz.
- „Eine Datei, viele Schreiber“ (SFMW; „Single File Multiple Writer“)): Jeder Prozess nimmt an einem gemeinsamen parallelen Schreibvorgang teil, der in eine einzelne Datei füllt.
- „Eine Datei pro Schreiber“ (FPP; „File Per Process“)): Wie bei SFMW schreiben alle Prozesse, aber jeder davon öffnet eine eigene Datei, die nur mit den eigenen Daten befüllt wird. Dieser Ansatz wird empfohlen, wenn kein äußerer Umstand dagegen spricht. [15]
- „Eine Datei, mehrere Schreiber“ (SFCW; „Single File Collective Writer“): Wie SFMW, aber nicht alle, sondern nur eine Teilmenge der Prozesse schreiben.

Diese Ansätze unterscheiden sich in der benötigten Prozesskommunikation. So ist es für SFSW nötig, dass alle Daten an den gleichen Prozess übertragen werden. Bei SFMW und FPP dagegen muss keine weitere Prozesskommunikation stattfinden. SFCW benötigt ein auf die Anzahl der Schreiber angepasstes Kommunikationsschema.

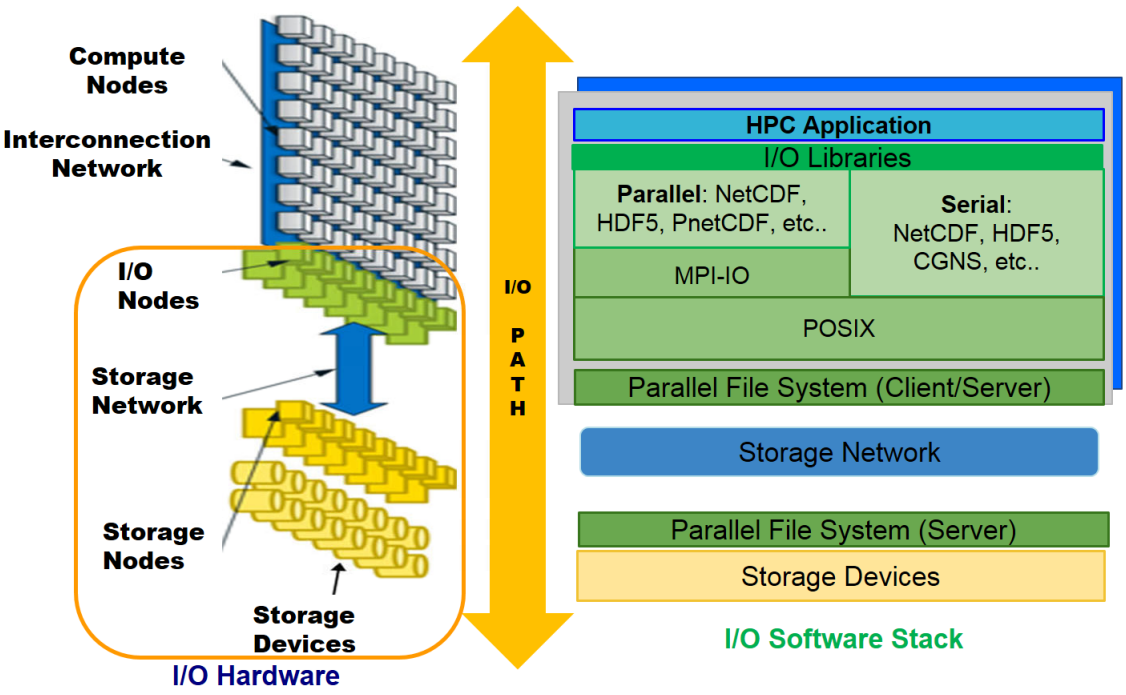

Abbildung 2.5.: Graphische Darstellung des E/A-Stacks eines modernen Supercomputers. [15]

Die Performanz dieser Ansätze kann nicht grundsätzlich unterschieden werden, da sie von der Umgebung abhängt. Bei einem nicht parallelen Dateisystem kann SFSW am schnellsten sein, da die verschiedenen Zugriffe der anderen Ansätze einen Mehraufwand erzeugen können. Andererseits können sie bei einem korrekt konfigurierten parallelen Dateisystem theoretisch einen Speedup gemäß Amdahls Gesetz erlauben.

2.2. Standards und Software

2.2.1. POSIX-Threads

POSIX-Threads sind eine Spezifikation für Threads, die im POSIX.1c-Standard definiert wird. Sie wird durch die Softwarebibliothek `phtreads` [25] implementiert. In ihr sind Funktionen zur Verwaltung von Threads definiert. Dazu gehören unter anderem die Möglichkeiten, Threads zu starten und eine bestimmte Funktion im Programm ausführen zu lassen, über sogenannte Pipes mit diesen Threads zu kommunizieren, Threads zu beenden, und vieles weiteres.

Ein neuer Thread wird mithilfe von `pthread_create` erzeugt. Dabei werden mehrere Argumente übergeben. Attribute bestimmen das Scheduling des gestarteten Threads und andere Meta-Informationen. Ein Funktionszeiger bestimmt die zu startende Funktion, Es

wird ein Argument für die zu startende Funktion als Zeiger übergeben. Mit dem `thread`-Zeiger wird eine ID zurückgegeben, die von da an den erstellten Thread referenziert.

```
1 int pthread_create(pthread_t *restrict thread,
2                    const pthread_attr_t *restrict attr,
3                    void *(*start_routine)(void *),
4                    void *restrict arg);
```

Listing 2.1: Signatur der `pthread_create`-Methode. [25]

Die Funktion `pthread_join` bildet das Gegenstück und wartet, bis der Thread mit der gegebenen ID abgeschlossen ist. Daraufhin gibt sie in dem Argument `retval` den Rückgabewert der Thread-Funktion zurück.

```
1 int pthread_join(pthread_t thread, void **retval);
```

Listing 2.2: Signatur der `pthread_join`-Methode. [25]

Es gibt außerdem noch Funktionen zur Prozesskommunikation, wie zum Beispiel `pthread_barrier_wait`. Diese Funktion nimmt einen Zeiger zu einer `pthread barrier_t` entgegen. Die Barriere gibt an, wie viele Prozesse gleichzeitig an der Barriere warten müssen. Die Funktion wartet, bis diese Zahl erreicht ist, und kehrt dann zurück. Damit lässt sich Prozesssynchronisation umsetzen.

```
1 int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Listing 2.3: Signatur der `pthread_barrier_wait`-Methode. [25]

2.2.2. OpenMP

Die OpenMP-API [21] ist eine Schnittstellendefinition, die eine Grundlage für portable Parallelisierung in C, C++ und Fortran bietet. Sie unterstützt eine Vielzahl an Konzepten, von Tasks über Arbeitsspeicherverwaltung bis hin zu Schleifentransformation. Insbesondere zeichnet sie sich durch ihre eher deklarative Natur aus. OpenMP-Direktiven geben oft an, welcher Natur die beschriebenen Konstrukte sind, und der Compiler baut daraus geeignete Parallelisierung. Ein Beispiel dafür ist die Deklaration `#pragma parallel for`, welche eine for-Schleife als parallelisierbar deklariert und ausreicht, um eine Parallelisierung zu erreichen.

Die Parallelisierung erfolgt dabei nach dem Fork-Join-Modell. Ein OpenMP-verwendendes Programm verwaltet automatisch die dafür nötigen Threads.

2.2.3. MPI

Das Message-Passing-Interface, kurz MPI, ist ein Standard für parallele Prozesskommunikation, -verwaltung, und viele weitere damit verbundene Themen. Es wurde ursprünglich 1994 entwickelt. 2021 wurde Version 4 veröffentlicht [16].

MPI umfasst eine Schnittstelle, die Funktionen für Kommunikation zwischen und Verwaltung von Prozessen während der Ausführungszeit bereitstellt, und eine Reihe an Hilfsprogrammen zur Konfiguration einer MPI-Umgebung und zur Kompilation und Ausführung von MPI-verwendenden Programmen. Ein MPI-Programm muss mit einem MPI-Compiler kompiliert werden, um die MPI-Schnittstelle nutzen zu können, und mit einem entsprechenden Befehl ausgeführt werden. Dieser Befehl startet dann anhand der Konfiguration mehrere Prozesse, die alle das gleiche Programm von Anfang an ausführen.

Aus Sicht des Anwenders der Schnittstelle wird die Komplexität der Umgebung abstrahiert. Prozesse werden in sogenannten Kommunikatoren gruppiert und können innerhalb eines Kommunikators miteinander kommunizieren. Am Anfang der Ausführung sind alle Prozesse in dem Kommunikator `MPI_COMM_WORLD` zusammengefasst. Ein Prozess kann Teil von mehr als einem Kommunikator sein. Innerhalb jedes Kommunikators sind die Prozesse nummeriert. Die Nummer eines Prozesses wird als `rank` bezeichnet, und die Gesamtgröße als `size`. Jedem Prozess ist der eigene Rang und die Größe innerhalb eines Kommunikators zugänglich.

Es werden verschiedene Funktionen bereitgestellt, die entweder kollektiv oder unabhängig sind. Eine kollektive Funktion muss von allen Prozessen eines Kommunikators aufgerufen werden. Dabei müssen oft mehrere der Argumente zwischen den Prozessen identisch sein. Sie umfassen solche Funktionen wie `MPI_Barrier`, die blockiert bis alle Prozesse sie aufgerufen haben und damit die Ausführung zwischen den Prozessen synchronisieren kann, oder `MPI_Allgather`, welche Daten zwischen allen Prozessen austauscht. Unabhängige Funktionen werden nur von einem Teil, manchmal sogar nur einzelnen Prozessen aufgerufen. Beispiele sind `MPI_Send` und `MPI_Recv`, die gemeinsam Punkt-zu-Punkt-Kommunikation ermöglichen. Zwei Prozesse können sie unabhängig von allen anderen Prozessen des Kommunikators aufrufen, um Daten miteinander auszutauschen. [1]

Ein Teil der Funktionen beschäftigt sich mit der Verwaltung von Kommunikatoren. Es können Subsets der Prozesse zu neuen Kommunikatoren zusammengefasst werden. Dadurch können zum Beispiel zwei Sets an Prozessen unterschiedliche Berechnungen ausführen, nur jeweils miteinander kommunizieren ohne von dem anderen Set beachtet werden zu müssen, und anschließend wieder gemeinsam Daten austauschen.

MPI umfasst noch viele weitere Bereiche und Themen. Die hier gegebene Übersicht ist allerdings ausreichend für das Verständnis der vorliegenden Arbeit.

Es gibt zwei große MPI-Implementationen, OpenMPI [20] und MPICH [17]. OpenMPI implementiert bisher nur MPI Version 3, während MPICH bereits Version 4 implementiert. Beide Projekte unterscheiden sich in Implementationsdetails und in unterstützter Hardware. Solange aber nur Funktionalität der Version 3 verwendet wird, ist diese Unterschied eher für Administratoren als für Programmierer interessant.

2.2.4. HDF5

HDF [30] ist sowohl ein Datenmodel, ein Dateiformat als auch eine Softwarebibliothek, um dieses Format zu schreiben. Das Dateiformat wurde in den neunziger Jahren als HDF4 von dem National Center for Supercomputing Applications (NSCA) als Standard

für wissenschaftlichen Datenaustausch festgelegt. Version 4 wird bis heute unterstützt, wurde aber auch zu HDF5 weiterentwickelt, welches 1998 veröffentlicht wurde. Es setzt sich als Ziel, wissenschaftliche Daten selbstbeschreibend und kompakt abzuspeichern. Die Biblothek ist auf flexible, effiziente, hochperformante Ein- und Ausgabe ausgelegt.

Eine HDF5-Datei wird durch einen gerichteten Graphen beschrieben, in dem Objekte die Knoten bilden. Diese Objekte sind entweder Datensets oder Gruppen. Jede Kante in dem Graphen hat einen Namen. Wenn eine Gruppe mit einem Namen bezeichnet wird, ist gemeint, dass eine Kante mit diesem Namen auf die Gruppe zeigt. Es existiert immer eine Wurzel-Gruppe mit dem Namen „/“ als Einstiegspunkt in die Datei. Sie stellt einen Sonderfall dar. Gruppen können auf andere Gruppen zeigen und somit eine hierarchische Datenstruktur bilden.

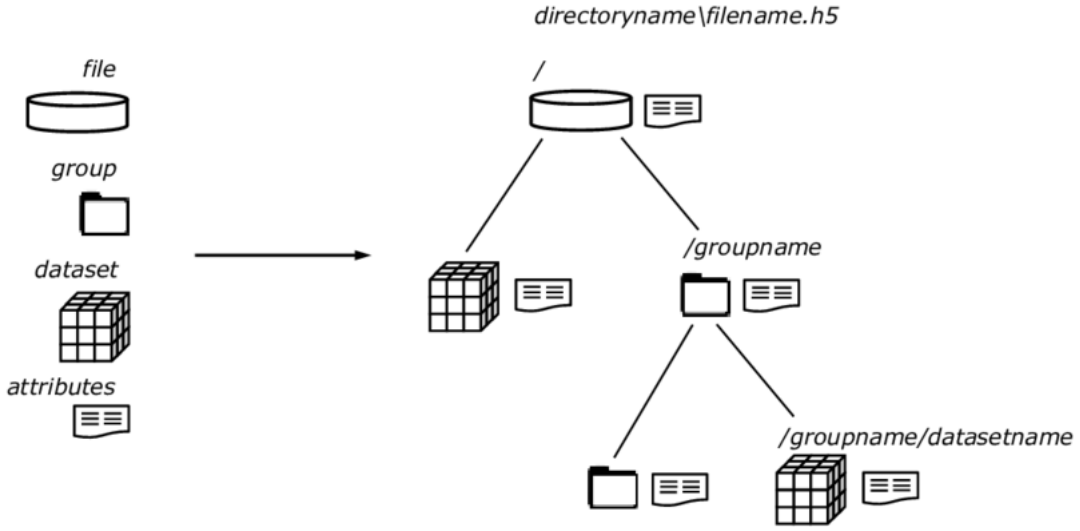

Abbildung 2.6.: Illustration des Aufbaus einer HDF5-Datei. [23]

Außerdem zeigen Gruppen auf Datensets, die Daten und diese Daten beschreibenden Metadaten beinhalten. Die Metadaten bestehen aus einem Datenraum, der die Dimensionen der Daten angibt, und einem Datentypen, der die Form der einzelnen Datenpunkte angibt.

Alle Objekte können mit Attributen annotiert werden. Sie haben eine ähnliche Struktur wie Datensets und sind für einfache Extrainformationen wie Parameter der dargestellten Messung vorgesehen.

Der Aufbau ist in Abbildung 2.6 illustriert.

Die dazugehörige Bibliothek besteht aus mehreren Modulen, die sich jeweils mit einem Aspekt des Formats beschäftigen. So ist z.B. das Modul `H5G` für Gruppen verantwortlich, während das Modul `H5D` Datensets manipuliert. Diese Module definieren kleinteilige Funktionen wie das Erschaffen und Freigeben eines korrespondieren Objekts, oder das Setzen einzelner Eigenschaften.

Zusätzlich unterstützt die Bibliothek in einem MPI-Kontext parallelen Zugriff auf Dateien. Dabei können Dateien entweder kollektiv bearbeitet werden, d.h. alle teilnehmenden Prozesse führen die Operationen gleichzeitig aus, oder unabhängig, bei der Prozesse nicht die gleichen Operationen ausführen müssen.

Der Zugriff auf HDF5-Daten erfordert die Definition von sogenannten Hyperslabs, welche einen Bereich der Daten auswählen. Sie bestehen aus:

- Einem Offset, der angibt, wo in der Datei die Auswahl beginnt.
- Einer Größe, die angibt, wie viele Daten eingelesen werden.
- Einer Schrittgröße (engl. *Stride*), die angibt, wie viele Schritte zwischen jedem gewählten Schritt liegen.
- Einer Blockgröße, die angibt, wie viele des definierten Hyperslabs gelesen werden sollen.

Hyperslabs können zusätzlich mit Set-Operationen modifiziert werden. Zum Beispiel kann eine komplexe Region als Vereinigung zweier Hyperslabs angegeben werden.

2.2.5. NetCDF

NetCDF [19] ist eine Software-Bibliothek, die CDF- und netCDF-Dateien liest und schreibt. Sie setzt sich als Ziel, portierbar und skalierbar selbstbeschreibende, erweiterbare und archivierbare Daten zu schreiben. Sie unterstützt parallelen Zugriff mit MPI. Dafür muss entweder die HDF5-Bibliothek oder die pnetcdf-Bibliothek vorhanden und in die Version von netCDF integriert sein. Sie erlauben parallelen Zugriff für netCDF-4- und CDF-5-Dateien respektive.

Die unterstützten Dateiformate sind CDF-1, CDF-2, netCDF-4, netCDF-4-Classic und CDF-5. NetCDF-4 ist kompatibel mit HDF5, während die CDF-Formate ein eigenständiges Format bilden.

Eine NetCDF-kompatible Datei besteht aus einer Reihe an Variablen, die Daten in als mehrdimensionales Array speichern, und aus Metadaten, die diese Daten beschreiben. Die Form jeder Variable wird als Kombination aus Dimensionen angegeben.

Jede Dimension besteht aus einem Namen und einer Größe, die angibt, wie viele Elemente in dieser Dimension enthalten sind. Die Größe kann als `NC_UNLIMITED` gegeben werden, um eine unendlich große Dimension zu definieren. Jede Variable darf mit genau einer unendliche Dimension definiert werden, die an erster Stelle stehen muss. Die Größe der Variable wird automatisch erweitert, wenn ein Wert an eine Stelle hinter der bisherigen Größe entlang dieser Dimension geschrieben wird.

Zusätzlich unterstützt netCDF-4 Gruppen, die analog zu denen aus HDF5 definiert sind.

Zugriff auf NetCDF-Daten erfordert eine Hyperslabs analog zu den in 2.2.4 erklärten.

3. Entwurf

In diesem Kapitel werden die grundlegenden Entwurfsziele definiert. Anhand dieser wird ein Beispielproblem ausgewählt und die Umsetzung dieses dargelegt. Die für die Parallelisierung relevante Entscheidungen werden hervorgehoben und begründet. Alternativen und ihre Vor- und Nachteile werden erläutert.

3.1. Zielsetzung

Ziel ist es, ein Programm zu entwickeln, dass als Einführung in das Feld der Parallelisierung dienen kann.

Dafür müssen die Parallelisierungsansätze Posix-Threads, OpenMP und MPI verwendet werden können und zu einem Speedup führen. Innerhalb dieser Anforderung ist es hilfreich, wenn die Ansätze am gleichen Punkt ansetzen. Durch die Vergleichbarkeit wird es einfacher, in der Verwendung der Software als Lehrmaterial Wissen zwischen den Ansätzen zu übertragen.

Die Stärken von NetCDF und HDF5 lassen sich nur zeigen, wenn das Programm durch die Menge der Ausgabedaten Ausgabe-gebunden wird. Entsprechend muss es möglich sein, viele speicherbare Daten zu erzeugen. Gleichzeitig sollte diese Ausgabe-Gebundenheit nicht verhindern, dass der Speedup der parallelen Berechnung demonstriert werden kann. Im Bestfall ist es möglich, dynamisch einem Ausgabe-gebundenen und einem CPU-gebundenen Modus auszuwählen.

Das Programm sollte nachvollziehbar sein. Um die Parallelisierung des Programms zu verstehen, muss auch das Programm selber verstanden werden können. Dazu zählt auch, dass die Ein- und Ausgabe des Programms verständlich ist, d.h. es ist für einen Einsteiger zumindest für einen Teil der möglichen Eingaben nachvollziehbar, wie die Ausgabe mit der Eingabe zusammenhängt.

Das Programm sollte möglichst testbar sein. Das hilft sowohl für die initiale Entwicklung als auch für die Verwendung als Lehrmaterial. Sie erlaubt die Implementation von Tests, um die Korrektheit des Programms vor und nach der Parallelisierung sicherzustellen. Zusätzlich erlaubt sie die Umsetzung von automatisierten Korrekturskripten u.ä. in der Lehre.

Daraus ergibt sich unter anderem, dass die Ausgabe des Programms möglichst stabil sein sollte. Gemeint ist damit, dass die Ausgabe für eine gegebene Eingabe nicht bzw. so wenig wie möglich von äußeren Umständen abhängt. Insbesondere gilt das für die eingesetzte Parallelisierung. Weder der Parallelisierungsansatz noch die Anzahl der verwendeten Threads/Prozesse sollte die Ausgabe beeinflussen.

Das Programm sollte einfach zu handhaben sein. Es sollte einfach sein, den Quellcode

zu kompilieren, das Programm auszuführen, die Ergebnisse auszuwerten und Änderungen am Programm vorzunehmen. Dazu zählt auch, dass die für die Ausführung notwendigen Eingabedaten eine gewisse Portabilität aufweisen sollten, entweder indem sie klein genug sind, um mit dem Quellcode mitgeliefert zu werden, oder indem sie Anwenderseitig erzeugbar sind.

Das Programm sollte eine möglichst hohe Codequalität aufweisen, unter anderem als tangentiale Eigenschaft der einfachen Handhabung. Codequalität hat einen Einfluss auf die Kosten eines Softwareprojektes. [6] Gerade im Kontext wissenschaftlicher Arbeit, in der die verfügbare Zeit und das Budget oft begrenzt sind, ist es also wichtig, die Codequalität im Rahmen der Möglichkeiten zu optimieren. Als Basis wurde der ISO-Standard für Softwarequalität herangezogen. Er definiert Wartbarkeit an den Begriffen Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit, und Konformität. Außerdem wird Übertragbarkeit an den Begriffen Anpassbarkeit, Installierbarkeit, Koexistenz, Austauschbarkeit, und wieder Konformität definiert. [29] Insbesondere die Übertragbarkeit trägt in diesem Projekt dazu bei, dass die Ergebnissen auch für andere Projekte relevant sind. Eine vollständige Evaluation ist jedoch zu umfassend, um für diese Arbeit durchgeführt zu werden. Daher war zwar die Entwicklung an diesen Qualitätsbegriffen orientiert, es wurde aber keine formelle Analyse durchgeführt.

Zusammengefasst ergeben sich diese Kriterien:

1. Ansatzpunkte für alle Parallelisierungen
2. Ausgabe- und CPU-Gebundenheit
3. Möglichst einfache Umsetzung
4. Testbarkeit
5. Stabile Ausgabe
6. Einfache Handhabung
7. Hohe Codequalität

3.2. Problemwahl

Es gibt viele Problemstellungen, an denen sich Parallelisierung erklären lässt, von Sortieralgorithmen über Phyiksimulationen bis hin zu Proteinanalyse. Um aus diesen eine als Gegenstand dieser Arbeit auszuwählen wurden die relevanten Kriterien herangezogen.

Diese sind Kriterium eins, zwei, drei, fünf und sechs. Das Kriterium der Testbarkeit und der Codequalität sind erst für die konkrete Umsetzung relevant.

Es werden sowohl das gewählte Problem, die N-Körper-Simulation, als auch beispielhaft zwei weitere, nicht gewählte Probleme analysiert. Anhand der Beispiele wird aufgezeigt, wann die Kriterien nicht erfüllt sein können.

3.2.1. Diskrete Fourier-Transformation

Ansatzpunkte	
POSIX-Threads	✓
OpenMP	✓
MPI	✓
Parallele Ausgabe	
Weitere Kriterien	
Ausgabe- und CPU-Gebundenheit	
Einfache Umsetzung	
Stabile Ausgabe	✓
Einfache Handhabung	✓

Tabelle 3.1.: Zusammenfassung der Kriterien für die diskrete Fourier-Transformation

Eine Fourier-Transformation zerlegt ein aperiodisches Signal in ein kontinuierliches Frequenzspektrum. Im Bereich der digitalen Signalverarbeitung spielt die sogenannte *Schnelle Fourier-Transformation* eine wichtige Rolle. Es existieren eine Vielzahl von Algorithmen. Insbesondere wird hier Cooley-Tukey-Algorithmus betrachetet, welcher den Yales-Algorithmus auf Fourier-Transformationen anwendet und damit eine parallele Berechnung ebendieser ermöglicht. [2].

Seien $x_0 \dots x_(N-1)$ eine Reihe komplexer Zahlen. Die diskrete Fourier-Transformation ist dann gegeben als Gleichung 3.1.

$$X_k = \sum_{j=0}^{N-1} x_j e^{-2\pi i \frac{jk}{N}}, \quad k = 0, \dots, N-1 \tag{3.1}$$

Für Cooley-Tukey werden die Indizes der Eingabereihe in zwei Reihen zerlegt, sodass diese parallel zueinander verarbeitet werden können. Daraus folgt, dass OpenMP idiomatisch anwendbar ist. Damit ist auch MPI und POSIX-Threads anwendbar. Die Größe der Ausgabe ist gleich der Größe der Eingabe. Das macht eine Berechnung mit einer großen Datenausgabe unpraktisch. Der Algorithmus ist damit ungeeignet, Ansätze der parallele Ausgabe zu illustrieren.

3.2.2. Neuronale Netzwerke

Neuronale Netzwerke erlauben die Approximation von beliebigen Funktionen durch einen automatisierten Lernprozess. Sie bestehen aus mehreren Schichten an „Neuronen“, welche jedes Eingabesignal der vorherigen Schicht mit einem Gewicht multiplizieren und dann eine sogenannte Aktivierungsfunktion auf das Ergebnis anwenden. Ein häufiges Anwendungsfeld ist die Klassifizierung von Bildern. Die Verarbeitung kann entlang sowohl entlang der Daten als auch entlang der Neuronen parallelisiert werden.

Ansatzpunkte	
POSIX-Threads	✓
OpenMP	✓
MPI	✓
Parallele Ausgabe	
Weitere Kriterien	
Ausgabe- und CPU-Gebundenheit	
Einfache Umsetzung	✓
Stabile Ausgabe	
Einfache Handhabung	

Tabelle 3.2.: Zusammenfassung der Kriterien für Neuronale Netzwerke

Dieses Problemfeld ist aus mehreren Gründen ungeeignet. Es werden viele Daten benötigt, um einen realistischen Fall abzubilden. Außerdem muss, wenn das trainieren des Netzes nicht der Zweck des Programms ist, ein trainiertes Modell gegeben sein. Ein solches trainiertes Modell zu erstellen ist selber bereits ein Aufwändiges Projekt und kann alleinig den Großteil einer Bachelorarbeit beanspruchen. (Siehe z.B. Wirths Arbeit zur Messung der Top Quark Masse mithilfe eines Neuronalen Netzwerkes. [33])

Das Training kann als Zweck gewählt werden. Dann nimmt der Trainingsprozess einen signifikanten Teil der Berechnungszeit ein. Es gibt Methoden, diesen Prozess zu parallelisieren, beschrieben z.B. von Hedge und Usmani. [8] In diesem Fall ist das Ergebnis allerdings nicht mehr deterministisch. In der sequentiellen Ausführung wird die Anpassung des Netzwerkes in jedem Lernschritt durch alle vorherigen Lernschritte beeinflusst. In der parallelen Ausführung lernen die Ausführungsstränge unabhängig voneinander und ihre Ergebnisse werden hinterher zusammengesetzt. Dadurch liegt die in der in der sequentiellen Ausführung vorhandene Beeinflussung so nicht vor und das Ergebnis ist zwangsläufig nicht identisch.

Zusätzlich sind die Ergebnisse nicht sinnvoll von Hand verifizierbar. Die Ausgabe eines neuronalen Netzes lässt sich für eine bestimmte Eingabe nicht vorhersagen.

Damit sind neuronale Netzwerke ungeeignet, Parallelisierungsansätze zu illustrieren.

3.2.3. Das gewählte Problem: Das N-Körper-Problem

Das N-Körper-Problem behandelt die Gravitationskraft im Bezug auf diskrete Körper. Instanzen dieses Problems sind zum Beispiel die Berechnung der Bewegung von Planetensystemen oder der Flugbahn von Asteroiden. Grundsätzlich müssen für das N-Körper-Problem die Bewegung von und die Gravitationskraft zwischen n Körpern berechnet werden.

Für eine geeignet große Anzahl an Körpern bietet dieses Problem einen ausgezeichneten Ansatzpunkt für alle Parallelisierungen. Bei einer gleichmäßigen Aufteilung der Körper haben alle teilnehmenden Ausführungsstränge eine ähnliche Last. Da die Positio-

Ansatzpunkte	✓
POSIX-Threads	✓
OpenMP	✓
MPI	✓
Parallele Ausgabe	✓
Weitere Kriterien	✓
Ausgabe- und CPU-Gebundenheit	✓
Einfache Umsetzung	✓
Stabile Ausgabe	✓
Einfache Handhabung	✓

Tabelle 3.3.: Zusammenfassung der Kriterien für das N-Körper-Problem

nen aller Körper in jedem Zeitschritt gebraucht werden, müssen alle Ausführungsstränge miteinander kommunizieren.

Es werden pro Zeitschritt Daten errechnet, die ausgegeben werden können. Jeder Ausführungsstrang erzeugt dabei Daten, die unabhängig voneinander ausgegeben werden können. Damit ist parallele Ausgabe sinnvoll anwendbar. Die Ausgabe kann sowohl die Positionsdaten, als auch die berechneten Kräfte umfassen. Die Positionsdaten haben dabei die Platzkomplexität $O(n)$ pro Zeiteinheit und die Kräfte eine von $O(n^2)$. Indem die Kraftausgabe ein- und ausgeschaltet wird kann somit die Platzkomplexität der Ausgabedaten angepasst werden, was bei einer genügend großen Eingabe zu einer Ausgabegebundenheit führen könnte.

Die Eingabedaten sind Positions-, Geschwindigkeits- und Massenangaben. Diese sind für einen Menschen nachvollziehbar. Für geringe Körperzahlen sind die Berechnungen auch händisch umsetzbar und können damit auch von Menschen verifiziert und nachvollzogen werden. Größere Fehler in der Berechnung können als solche erkannt werden, wenn zum Beispiel die Größenordnung der Ausgabedaten falsch ist. Die Ausgabedaten sind deterministisch. Sobald eine Umsetzung der Simulation händisch als korrekt identifiziert wurde, lässt sich dessen Ergebnis als Basis eines Regressionstests verwenden, um bei Änderungen an der Simulation die Korrektheit zu verifizieren.

Damit erfüllt das N-Körper-Problem alle Kriterien und wird als Beispielproblem verwendet.

3.3. Komponenten des Programs

Das Programm wird in verschiedene Komponenten aufgeteilt, die jeweils eine spezifische Aufgabe haben. Diese Komponenten werden hier kurz erläutert.

utility

Die Komponente `utility`, bestehend aus `utility.h` und `utility.c`, beinhaltet Hilfsdefinitionen. Insbesondere zählen dazu die Funktionen, die zur Berechnung der Datenaufteilung (siehe 3.6.1 verwendet werden.

vector3

Die Komponente `vector3`, bestehend aus `vector3.h` und `vector3.c`, beinhaltet eine Definition für einen 3-dimensionalen Vektor und Funktionen, um mit diesen zu rechnen. Diese Funktionen sind Einheiten- und Größenunabhängig. Das heißt, dass der verwendende Code darauf achten muss, die physikalischen Größen richtig zu verwenden. Bei den implementierten Funktionen handelt es sich um einfache mathematische Operationen wie das Addieren von zwei Vektoren, oder die Distanz zwischen zwei durch Vektoren beschriebenen Punkten zu finden.

Außerdem beinhaltet dieser Code die Definition für eine Reduktionsoperation auf Vektoren für OpenMP. Diese wird in 3.6.3 erklärt.

options

Die Komponente `options`, bestehend aus `options.h` und `options.c`, implementiert eine GNU-ähnliche Kommandozeilenschnittstelle. Es stellt Optionen bereit, die Berechnungsparameter wie die Größe der Zeitschritte und die Menge der durchgeführten Schritte zu definieren, einzelne Körper für die Simulation anzugeben, oder eine Datei zum Einlesen einer ganzen Liste an Körpern zu definieren. Ein Körper wird dabei als Liste aus Masse, Orts- und Geschwindigkeitsangaben angegeben. Außerdem erlaubt es, die Ausgabe zu konfigurieren, indem die Ausgabestrategie und das Ausgabeformat angegeben werden.

Diese Optionen werden eingelesen und als Datenstruktur aus direkt verwendbaren Typen an das Hauptprogramm weitergegeben.

io

Die Komponente `io` besteht aus mehreren Header- und Source-Dateien. Die Interface-Dateien `io.h` und `io.c` bieten eine abstrahierte IO-Schnittstelle und eine Funktion, um die Verwendung dieser zu konfigurieren. Die bereitgestellten Ausgabemöglichkeiten sind wiederum in dem Unterordner `io` definiert und sollten nicht direkt verwendet werden.

Die bereitgestellten Ausgabeschnittstellen werden in 3.7 erläutert.

Dieser Aufbau ermöglicht, den zentralen Algorithmus Schnittstellenunabhängig zu formulieren und die Komplexität der IO-Implementierung vom Algorithmus zu trennen. Dadurch ist es einfacher, den jeweiligen Code nachzuvollziehen und gegebenenfalls frei von Nebeneffekten anzupassen.

nbody

Die Komponente `nbody`, bestehend aus `nbody.h` und `nbody.c`, implementiert den zentralen Algorithmus wie in 3.4 beschrieben und setzt die anderen Komponenten dafür zusammen. Sie implementiert die nötigen Berechnungsschritte und die nötigen Parallelisierungsmechanismen, die sich nicht auf IO beziehen.

3.4. Zentraler Algorithmus

Eine N-Körper-Simulation kann auf viele Arten und Weisen umgesetzt werden, die verschieden Komplex in ihrer Implementation, ihren Berechnungen und ihrer Parallelisierung sind. Ein Beispiel ist die Barnes-Hut-Simulation [35], bei der zu simulierende Bereich in einzelne Zellen unterteilt wird. Exakte Berechnungen werden dann nur für benachbarte Zellen durchgeführt, während die Objekte weiter entfernter Zellen zusammengefasst werden. Dabei können dann verschiedene Zellen verschiedenen Prozessen zugeordnet werden.

Eine wichtige Frage ist auch, ob die spezielle Relativitätstheorie angewendet wird, ob die Simulation sich auf Planetensystem nach den Kepplerschen Gesetzen beschränkt oder ob nur das Newtonsche Gravitationsgesetz betrachtet wird. In dieser Arbeit steht die Parallelisierung des gewählten Verfahrens im Mittelpunkt und die Präzision der Ergebnisse ist zweitrangig. Auch ist die Performanz des zugrundeliegenden Algorithmus nicht Fokus der Arbeit, solange sich die Möglichkeiten von Parallelisierung zeigen lassen.

Damit fällt die Wahl auf eine kontinuierliche Simulation anhand des Newtonschen Gravitationsgesetzes. Diese Art von Simulation folgt einem einfach zu verstehenden und perfekt parallelisierbaren Schema. Für jeden Körper wird die Summe der einwirkenden Kräfte aller anderen Körper berechnet. Dieser Kraftvektor wird auf die aktuelle Geschwindigkeit angewendet und der Körper wird entsprechend der berechneten Geschwindigkeit und dem gewählten Zeitschritt bewegt. Diese Vorgehensweise wird in Abbildung 3.1 illustriert.

Zusätzlich wird das Zentrum des Körpersystems berechnet und das Gesamtsystem rezentriert, sodass sich dieses Zentrum bei der Koordinate (0, 0, 0) befindet. Dies stellt sicher, dass die Werte der Positionsvektoren im Falle eines sich bewegendes Gesamtsystems nicht zu groß für eine genügend akkurate Fließkommadarstellung werden.

Die Berechnung kann mit einem Speicherverbrauch $O(n)$ durchgeführt werden, wenn die Teilschritte klar getrennt werden. Dann wird in einer ersten Schleife für jeden Körper die auf ihn wirkende Kraft berechnet und auf die Geschwindigkeit angewendet, und dann in einer zweiten Schleife die neue Position berechnet.

Nach jedem Zeitschritt werden die berechneten Daten ausgegeben. Diese umfassen die neuen Positionen und Geschwindigkeiten. Die Massen werden auch ausgegeben, aber nur einmal am Anfang der Simulation. Da sie nicht verändert werden ist eine mehrfache Ausgabe nicht notwendig.

Wenn zusätzlich die berechneten Kräfte ausgegeben werden sollen, müssen die berechneten Kräfte entweder laufend ausgegeben werden, oder die Kräfte mit einem Speicher-

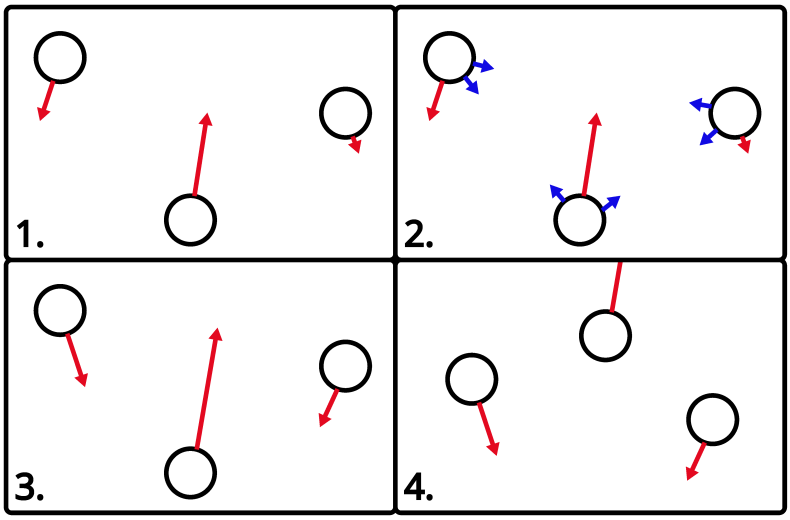

Abbildung 3.1.: Das N-Körper-Problem illustriert mit drei Körpern.
1. Der Ausgangszustand der Körper. Die roten Pfeile stellen ihre Geschwindigkeit dar.
2. Die Kräfte, dargestellt durch die blauen Pfeile, werden berechnet.
3. Die Geschwindigkeiten werden anhand der Kräfte angepasst.
4. Die Körper werden anhand der Geschwindigkeiten bewegt.

verbrauch von $O(n^2)$ (Eine Kraft pro Körper, pro Körper) zwischengespeichert werden. Letztere Methode erlaubt es, den gleichen Ausgabeschritt für die Positions- und die Kraftdaten zu verwenden und vermeidet es, in mitten von parallelen Ausführungen Ausgabe zu tätigen. Daher fiel die Wahl auf die letztere Methode.

Die Kraft wird von jedem Körper zu jedem anderen Körper berechnet, womit die Zeitkomplexität bei diesem Vorgehen bei $O(n^2)$ liegt. Es ist zwar möglich, die Hälfte der Berechnungen zu sparen, indem die Berechnung zwischen jeweils zwei Körpern nur einmal ausgeführt wird, dies ändert jedoch nichts an der Zeitkomplexität und verkompliziert die Implementation des Algorithmus. Daher wurde für diese Optimierung verzichtet.

Da jeder Körper durch einen Positions- und Geschwindigkeitsvektor sowie einer Masse beschrieben wird, wirkt es sinnvoll eine Datenstruktur zu definieren, welche alle diese Daten pro Körper zusammenfasst. Dieses Vorgehen stellt sich allerdings als ungeeignet für auf Geschwindigkeit optimierte Ausführung (in C) heraus. Verschiedenste Funktionen erwarten ein Array an homogenen Daten und erlauben es nicht, nur die notwendigen Teile eines Arrays aus Datenstrukturen zu verwenden. Ein konkretes Beispiel dafür ist die Ausgabe der Daten mit NetCDF, beschrieben in Abschnitt 3.7.1.

Daher werden die Vektoren und die Massen jeweils in einzelnen Arrays geführt, und die Array-Zeiger werden in einer Datenstruktur zusammengefasst. Dieser Ansatz wird *Structure of Arrays* (SoA) genannt. [12] Dieser Ansatz macht die Aufteilung der Daten

in den parallelen Varianten einfacher, da die Positionsdaten in jedem Prozess gebraucht werden, die Geschwindigkeitsdaten aber nur dort, wo die nächste Position berechnet werden soll.

3.5. Regressionstests

Um die Korrektheit des Algorithmus sicherzustellen, werden Regressionstests eingesetzt. Sobald die sequentielle Variante das erste mal umgesetzt ist, werden die Ergebnisse von Hand verifiziert und anschließend als Vergleichsdaten gespeichert. Von da an bilden diese Vergleichsdaten die Basis für die weitere Entwicklung.

Es werden Tests implementiert, die die Ausgabe des Programms mit den Vergleichsdaten vergleicht. Bei Abweichungen schlagen die Tests fehl. Wenn ein neuer Testfall hinzukommt, werden die Tests ausgeführt und der Testfall von Hand verifiziert. Wenn keine Abweichungen festgestellt werden, wird der Testfall in die Regressionstests aufgenommen. Entsprechend des Determinismus-Kriteriums wird erwartet, dass auch die parallelen Umsetzungen die Regressionstests bestehen.

3.6. Parallele Implementationen

Es werden drei Varianten des Programms definiert und über Makrodefinitionen zugänglich gemacht. Diese sind `NBODY_POSIX`, `NBODY_OPENMP` und `NBODY_MPI`.

`POSIX` verwendet Posix-Threads, um die Berechnung zu parallelisieren. Die Ausgabe ist nicht parallelisiert.

`OpenMP` verwendet OpenMP, um die Berechnung zu parallelisieren. Die Ausgabe ist nicht parallelisiert.

`MPI` verwendet MPI, um die Berechnung und die Ausgabe zu parallelisieren. Dabei wird die Schnittstelle für die Berechnung und die Umsetzung von Ausgabeschemata, beschrieben in 3.7.2, direkt verwendet. Die Ausgabe selber wird durch die NetCDF- und die HDF5-Bibliothek mit MPI parallelisiert.

Variante	Macro	Parallele Berechnung	Parallele Ausgabe
POSIX	NBODY_POSIX	x	
OpenMP	NBODY_OPENMP	x	
MPI	NBODY_MPI	x	x

Tabelle 3.4.: Übersicht der Varianten

3.6.1. Datenaufteilung

Für die Parallelisierung müssen die Daten auf die Prozesse aufgeteilt werden. Die Datenmenge wird dafür in mehrere gleichmäßige Bereiche unterteilt. Eine gleichmäßige

Aufteilung sorgt für eine ähnliche Last zwischen den Prozessen, da für jedes Element die gleichen Operationen ausgeführt werden.

Sei $p : 0..n : n \in \mathbb{N}$ die Menge der Prozesse und $s : 0..n : n \in \mathbb{N}$ die Menge der zu schreibenden Elemente. Jeder Prozess $p_n : n \in p$ übernimmt die Berechnung für den Bereich $B(p_n)$. Jeder Bereich bekommt $\lfloor |s|/|p| \rfloor$ Elemente, mit Ausnahme der ersten $|s| \pmod{|}p|$ Bereiche, welche jeweils ein Element mehr bekommen. Damit sind $(s \pmod p) * p + \lfloor s/p \rfloor = s$ Elemente aufgeteilt.

Die Bereiche sind kontinuierlich: Sei $O(p_n)$ das größte Element, das gerade nicht in $B(p_n)$ enthalten ist und $U(p_n)$ das kleinste Element, das gerade in $B(p_n)$ enthalten ist.

$$\forall e \in s : U(p_n) \leq e < O(p_n) \Leftrightarrow e \in B(p_n)$$

Aus der Definition von $U(p)$ und $O(p)$ ergibt sich außerdem, dass $U(p+1) = O(p)$.

s	0 1 2	3 4 5	6 7 8	9 10	11 12	13 14	15 16	17 18
p	0	1	2	3	4	5	6	7

Abbildung 3.2.: Aufteilung der Daten auf die Prozesse mit $p = 8$ und $s = 19$

Das `utility`-Modul definiert eine Reihe an Hilfsfunktionen, um diese Datenaufteilung vorzunehmen.

3.6.2. POSIX-Threads

Posix-Threads werden mithilfe der Bibliothek `pthread` verwendet. Um Posix-Threads zu nutzen, muss beim Programmaufruf eine Zahl an Threads p als Kommandozeilenargument angegeben werden. Bei der Ausführung werden dann $p - 1$ Threads mit `pthread_create` erzeugt. Der primäre Thread nimmt auch an der parallelen Ausführung teil. Damit bestehen zu keinem Zeitpunkt mehr als p Threads.

Jeder Thread übernimmt einen Teil der Arbeit entsprechend der Datenaufteilung. Entsprechend der Signatur der `pthread`-Schnittstelle werden die für die Berechnung nötigen Argumente in einer Datenstruktur zusammengefasst und so an eine dafür definierten Funktion übergeben. Mithilfe von `pthread_join` wartet der primäre Thread auf den Abschluss der erzeugten Threads. Der Rückgabewert der Routine ist die für die Rezentrierung berechnete Distanz. Die Änderungen an den Zuständen der werden direkt in das Array der *Structure of Arrays* geschrieben und müssen damit nicht zurückgegeben werden.

Die berechnende Funktion wird nicht für die anderen Varianten wiederverwendet, da zusätzliche `malloc` und `free` Aufrufe nötig wären.

Es gibt die Möglichkeit, einmal am Anfang der Anwendung Arbeiter-Threads zu erzeugen und danach in der Ausführung mit Arbeit zu versorgen. Dieser Aufbau vermeidet ein häufiges erzeugen von Threads, ist aber gleichzeitig deutlich aufwändiger zu implementieren. Die Möglichkeit wurde nicht umgesetzt, um die Varianten vergleichbar zu halten.

3.6.3. OpenMP

Der Algorithmus beinhaltet für die Geschwindigkeits- und die Positionsberechnung jeweils eine Schleife, deren Schritte jeweils unabhängig voneinander sind. Damit lässt sich OpenMP direkt anwenden, indem sie mit entsprechenden pragmas versehen werden. Diese sind in 4.5 erklärt.

3.6.4. MPI

Die Verwendung von MPI erfordert, dass das Programm mehrfach parallel ausgeführt wird. Jeder dieser Ausführungen konfiguriert sich selber so, dass sie nur ihren Anteil an der Arbeit ausführt. Die Ergebnisse dieser Berechnungen, konkret die Positionsdaten und die für die Rezentrierung nötige Gesamtdistanz, werden nach dem Berechnungsschritt zwischen den Prozessen in einer synchronen kollektiven Operation kommuniziert. Damit haben alle Prozesse die für den nächsten Schritt notwendigen Daten, bevor sie mit diesem anfangen. Die Geschwindigkeitsdaten müssen nicht ausgetauscht werden, da jeder Prozess nur seine eigenen Körper bewegt und damit die Geschwindigkeiten der anderen Körper für die Berechnung nicht relevant sind.

Initialisierender Code, zum Beispiel der Code zum Verarbeiten der Kommandozeilenargumente, wird mehrfach ausgeführt. Prinzipiell könnte die Möglichkeit der Kommunikation zwischen den Prozessen dafür genutzt werden, das zu vermeiden. Es wurde sich dagegen entschieden, da das die Initialisierung komplexer machen würde und gleichzeitig wenig Vorteile zu erwarten sind. Der Ressourcenintensivste Teil der Initialisierung ist das einlesen von Dateien, die Körper angeben. Die Gesamtrechendauer aber skaliert quadratisch mit der Größe ebenjener Dateien. Sobald diese Dateien also groß genug sind, um einen messbaren Einfluss auf die Performanz zu haben, wird ihr Einfluss von der Gesamtrechenzeit überschattet. Damit ist eine solche Optimierung den Aufwand nicht wert.

3.7. Ausgabe

Die Ausgabe ist darauf ausgelegt, alle in 2.1.6 genannten Strategien und alle Ausgabeformate ohne Rekompilation zu ermöglichen. Zusätzlich soll sie möglichst unabhängig vom berechnenden Code ausgeführt werden können. Dafür bildet eine Schnittstelle eine dünne Schicht zwischen dem berechnenden Code und dem schreibenden Code. Der schreibende Code ist dabei pro Ausgabeformat implementiert und wird dynamisch ausgewählt und aufgerufen. Dafür wird eine `io_args`-Datenstruktur erzeugt, die von den anderen Funktionen der Schnittstelle verwendet wird. Sie beinhaltet alle Informationen, die für die Ausführung der gewählten Ausgabeparameter nötig sind.

Die Funktion `configure_io` übernimmt die Konfiguration der Schnittstelle. Dabei werden drei Schritte ausgeführt. Erst wird festgestellt, ob eine Ausgabe stattfinden soll. Dann wird der schreibende Code entsprechend des gewählten Ausgabeformats ausgewählt. Schließlich wird anhand der Ausgabestrategie ein Kommunikationsschema erstellt.

```
1  typedef struct {
2      file_type_e file_type;
3      io_strategy_e strategy;
4      int io_size;
5      char *file_name;
6  } io_options;
7
8  typedef struct {
9      bool enabled;
10     io_schema_t io_schema;
11     io_strategy_e strategy;
12     char * file_name;
13     file_info * file;
14     open_file_pointer *open_file;
15     write_file_pointer *write_file;
16     close_file_pointer *close_file;
17 } io_args;
18
19 io_args configure_io(int rank, int size, size_t body_count,
       io_options options);
20 void open_file(io_args * args, body_list_t initial_bodies,
       double timestep);
21 void write_file(io_args * args, body_list_t bodies, size_t
       step);
22 void close_file(io_args * args);
23 void free_io_args(io_args args);
```

Abbildung 3.3.: Definition der Ausgabeschnittstelle.

Für Schema werden effektiv zwei Fälle unterschieden. Für die Strategie FPP wird dynamisch ein Dateiname erzeugt und das Schema so konfiguriert, dass jeder Prozess alleine alle eigenen Daten schreibt. Die anderen drei Strategien (SFSW, SFCW, SFWM) lassen sich alle als Fälle von SFCW betrachten. Sei p_w die Menge der schreibenden Prozesse. Dann ist SFSW der Fall $|p_w| = 1$ und SFMW der Fall $|p_w| = |p|$. SFCW deck alle Fälle $1 < |p_w| < |p|$ ab. Das dafür nötige Kommunikationsschema wird in 3.7.2 beschrieben. Über die Argumente `rank` und `size` werden geben die Prozess-ID und die Gesamtzahl an Prozessen angegeben. Im sequentiellen Fall ist „rank“ immer 0 und „size“ immer 1.

Sowohl NetCDF als auch HDF5 unterstützen von sich aus nur parallele Ausgabe per MPI [18]. Daher wurde die parallele Ausgabe anhand dieses Schemas ist nur für die MPI-Variante des Programms implementiert.

Die Ausgabe erfolgt mithilfe von drei Funktionen, die sich an der üblichen Struktur von

Dateizugriffen orientieren. Die `open_file`-Funktion bereitet eine Datei vor und schreibt den initial eingelesenen Zustand. Sie wird direkt nach dem Einlesen der Parameter aufgerufen. Die `write_file`-Funktion schreibt einen gegebenen Zeitschritt in die Datei. Diese Funktion kann und wird mehrmals aufgerufen. Die `close_file`-Funktion nimmt alle notwendigen Schritte vor, um den Schreibvorgang abzuschließen, zum Beispiel die Freigabe von reservierten Ressourcen.

Die Schnittstelle wird für jede der Ausgabemöglichkeiten NetCDF, HFD5 und GIF implementiert. Die Implementation umfasst sowohl den jeweiligen Code, um eine entsprechende Datei zu schreiben, als auch die konkrete Umsetzung der Ausgabestrategien. Dabei wurde darauf geachtet, dass die Ausgabe unabhängig der gewählten Strategie identisch ist, um die gewünschte Vergleichbarkeit zu gewährleisten.

Es wurde in Betracht gezogen, die Daten nicht pro Zeitschritt zu schreiben, sondern einen Datensatz gewisser Größe zu puffern und erst dann zu schreiben, wenn eine größere Datenmenge gesammelt wurde. Dieser Ansatz würde den Speicherverbrauch erhöhen und den Code maßgeblich komplexer machen. Außerdem setzen einige Ausgabebibliotheken unter Umständen selber Buffering um (Bspw. [7]). Daher wurde diese Strategie nicht umgesetzt.

Zusätzlich wurde eine Ausgabe über die Standard-Datenströme umgesetzt, die mit einem Kommandozeilenargument aktiviert werden kann.

3.7.1. Ausgabeformate

Die Module für die verschiedenen Ausgabeformate teilen sich eine gemeinsame Schnittstelle, die den schreibenden Code von der Implementation der Ausgabestrategie trennt. Der schreibende Code kann damit darauf ausgelegt werden, einen einfachen Datenblock zu schreiben, und bleibt dabei übersichtlich und verständlich.

```
1 typedef file_info *(open_file_pointer)(int rank, int size,
2 #ifdef NBODY_MPI
3 MPI_Comm communicator,
4 #endif
5 char *file_name, body_list_t initial_bodies, double timestep
     );
6
7 typedef void (write_file_pointer)(int rank, int size,
     file_info *file, body_list_t bodies, size_t step);
8
9 typedef void (close_file_pointer)(int rank, int size,
     file_info *file);
```

Abbildung 3.4.: Definition der Schnittstelle der Ausgabeformate.

Diese Schnittstelle besteht aus drei Funktionszeiger-Definitionen. Die `open_file`-

Funktion erzeugt eine `file_info`, welche von den anderen Funktionen entgegengenommen wird. In ihr sind Informationen enthalten, die zu den jeweiligen Ausgabeformaten gehören und von den anderen Funktion gebraucht werden. Es handelt sich um einen Zeiger, damit jeder Ansatz eine eigene Datenstruktur definieren und übergeben kann - der Zeiger wird dann zwischen den jeweiligen Datentypen umgewandelt. Die `write_file`-Funktion schreibt die in `bodies` übergebenen Daten an den gegebenen Zeitschritt `step`. Es wird erwartet, dass `step` in jedem Aufruf größer als vorher ist. Die `close_file`-Funktion hat auch die Aufgabe, die durch `file_info` verwendeten Resourcen freizugeben.

HDF5

In der HDF5-Ausgabe werden zwei Datensets definiert, um die Struktur der zu schreibenden Daten am besten ausnutzen zu können. Durch diese Entscheidung verdoppeln sich die notwendigen Aufrufe, dafür sind die schreibenden Aufrufe einfacher.

Das eine Datenset beinhaltet die Positionsdaten, das andere die Geschwindigkeitsdaten. Sie heißen `nbody_position` und `nbody_velocity`. Sie teilen sich den gleichen Datenraum, der aus drei Dimensionen besteht. Die erste ist eine unendliche Zeitdimension, die wenn nötig erweitert wird. Die zweite Dimension hat die Größe $|s|$ und stellt eine Zuordnung von Werten zu Objekten dar, d.h. der Wert an Position n entspricht Objekt n. Die dritte und letzte Dimension hat die Größe 3. Es wird kein Datentyp für die Vektoren definiert, da es einfacher ist, sie als 3-dimensionale Double-Werte zu betrachten. Diese Betrachtung ist in der dritten Dimension abgebildet.

Zusätzlich werden Textattribute an die Wurzelgruppe angehängt, die kurz die Natur der Daten erklären. Diese Attribute dienen vor allem der Veranschaulichung der Möglichkeit, Daten in HDF5 zu annotieren, und haben keinen konkreten Nutzen im Rahmen des Programms. Die Massen der Körper und die Größe des Zeitschritts werden auch als Attribute an die Wurzelgruppe geschrieben. Die erzeugten IDs, die für den Schreibprozess noch gebraucht werden, werden in mithilfe der `file_info`-Datenstruktur erhalten. Sie werden im `close_file`-Schritt geschlossen. Mit schließen der Datei-ID wird auch der Schreibvorgang abgeschlossen.

Es wird Kompression für die Daten aktiviert, da die Umsetzung trivial ist.

NetCDF

In der NetCDF-Ausgabe werden mehrere Variablen definiert:

Die IDs dieser Variablen werden in der `file_info`-Datenstruktur weitergereicht.

Der Schreibvorgang erfolgt analog zu HDF5, um eine Vergleichbarkeit der Ansätze zu gewährleisten.

An der NetCDF-Bibliothek zeigt sich das Problem mit dem Objektorientierten Ansatz, das in 3.4 angesprochen wurde. Mit diesem Ansatz könnte die Funktion `nc_put_vara_double` nicht effizient angewendet werden. Diese Funktion sieht vor, das ein Array aus gleichen Werten in eine Variable geschrieben wird. In der Datenstruktur allerdings in Werte verschiedener Art hintereinander vor. Entsprechend muss jeder Block von Werten einzeln

Variable	Dimension	Beschreibung	Größe
`body_position`	`time`	Zeitschritt	Unendlich
	`body_id`	Körper-Zuordnung	$\|s\|$
	`position`	Positionsvektor	3
`body_velocity`	`time`	Zeitschritt	Unendlich
	`body_id`	Körper-Zuordnung	$\|s\|$
	`velocity`	Geschwindigkeitsvektor	3
`body_mass`	`body_id`	Körper-Zuordnung	$\|s\|$
	`mass`	Skalare Masse	1
`timestep`	`timestep`	Skalare Zeit	1

Tabelle 3.5.: Die für die NetCDF-Ausgabe definierten Variablen und ihre Dimensionen. Dimensionen mit gleichen Namen werden nur einmal definiert.

geschrieben werden, was den Mehraufwand der Schreibmethode kumuliert. Die Funktion `nc_put_varm_double` erlaubt es, dennoch in nur einem Aufruf alle Werte zu schreiben, indem in einem weiteren Parameter die Distanz zwischen den Werten im zu schreibenden Array angegeben werden. Da diese Funktion aber als `deprecated` markiert ist entspricht diese Lösung nicht der vorgesehenen Herangehensweise, und es existiert keine alternative Funktion, die dieses Problem löst. Zusätzlich zeigt sich in der Analyse mit Vampir, dass selbst mit Verwendung dieser nicht vorgesehenen Funktion die Performanz leidet, da sie im Hintergrund genau die Art von mehrfachen Schreibaufruf verursacht, der durch sie vermieden werden sollte.

3.7.2. Ausgabeschema

Basierend auf der Datenaufteilung in 3.6.1 wird ein Schema zur Aufteilung der Daten zur Ausgabe definiert.

Sei $p_w : 0..n : n \in \mathbb{N}, n \leq |p|$ die Menge der schreibenden Prozesse. B_w, U_w, O_w seien Analog zu B, U, O definiert, in Relation zu $|p_w|$ statt $|p|$. Es muss eine Zuordnung $W(p_n) : p_n \in p, W(p_n) \in p_w$ definiert werden, die jedem schreibenden Prozess $p_{wn} \in p_w$ einem Prozess $p_n \in p$ zuordnet.

Eine Möglichkeit wäre, die Prozesse danach auszuwählen, wie viele „ihrer“ Elemente Teil des korrespondierenden Schreibprozesses wären und somit die Menge an übertragenen Daten zu minimieren. Dabei kann es aber dazu kommen, das ein Prozess an zwei andere Prozesse Daten schicken muss. Gleichzeitig müsste es einen Entscheidungsmechanismus geben, falls zwei Prozesse einen gleich großen Anteil an den Daten des Ausgabeprozesses haben.

Stattdessen werden die Zuordnungen danach entschieden, in welchem Prozess der Datenbereich des Ausgabeprozesses anfängt. Da $|p_w \leq p| \Rightarrow |s|/|p_w| \geq |s|/|p|$ liegt in jedem Datenbereich eines Prozesses genau eine oder keine Grenze eines Ausgabedatenbereiches. Damit muss jeder Prozess, der kein Ausgabeprozess ist, alle seine Daten an den

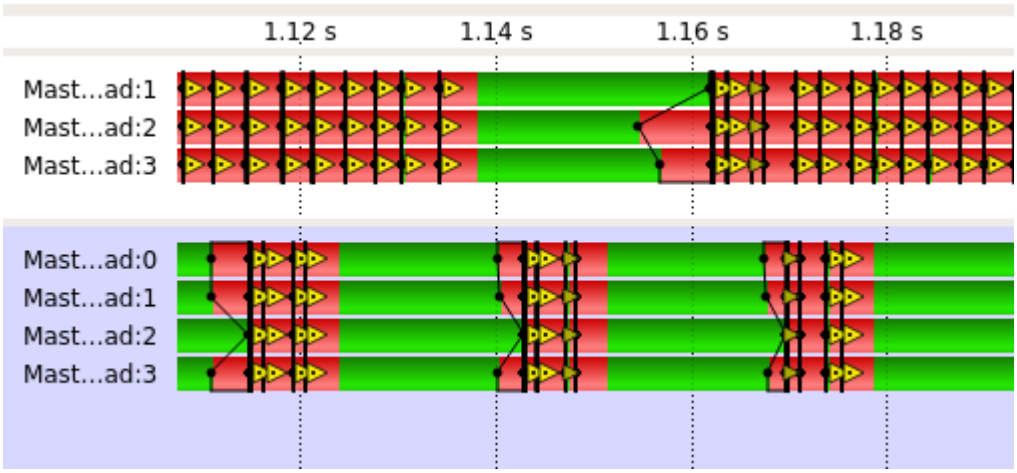

Abbildung 3.5.: Analyse des Zugriffsmusters mit Vampir. Oben ist das Zugriffsmusters der `varm`-Variante zu sehen. Rechts ist das Zugriffsmuster der `vara`-Variante zu sehen. Es ist eindeutig erkennbar, dass bei `varm` viele einzelne Ausgabeoperationen durchgeführt werden, während bei `vara` nach schon vier Ausgabeoperationen die nächste kollektive Operation stattfindet.

Ausgabeprozess mit überlappenden Datenbereich schicken. Jeder Ausgabeprozesse p_{wn} muss den Teil seiner Daten, der nicht in seinem Ausgabedatenbereich liegt, an den Ausgabeprozess $p_{wn} - 1$ senden. Da $B(0) = 0 \wedge B_w(0) = 0$ muss Prozess 0 niemals Daten schicken.

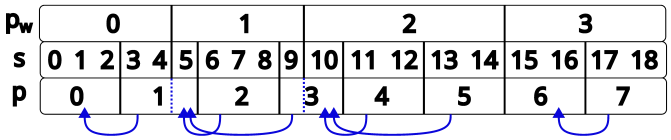

Abbildung 3.6.: Aufteilung der Daten auf die Prozesse und Ausgabeprozesse mit $p = 8$, $p_w = 4$ und $s = 19$. Die Pfeile veranschaulichen den nötigen Datentransfer.

Es werden höchstens $|p_w| - 1$ mal Daten transferiert, da jeder Prozess höchstens einen Datensatz transferiert und manche Prozesse (genau dann, wenn $B(p_n) = B_w(W(p_n))$) keine Daten transferieren müssen.

3.8. Schnittstelle

Das Programm wird über eine Kommandozeilenschnittstelle angesprochen. Mithilfe von Argumenten im Unix-Stil können die Parameter der Ausführung gesetzt werden. Die folgenden Argumente steuern die Simulationsparameter für alle Varianten des Programms:

- `timestep` (`t`): Gibt die Länge jedes einzeln Zeitschritts in Sekunden an. Wenn diese Option nicht angegeben wird, wird der Wert auf eine Sekunde gesetzt.
- `step-number` (`n`): Gibt die Anzahl der auszuführenden Schritte an. Muss angegeben werden, wenn `total_duration` nicht angegeben wird. Darf nicht gleichzeitig mit `total_duration` verwendet werden.
- `total_duration` (`d`): Gibt die Länge der Zeitspanne an, die simuliert werden soll. Die Anzahl der Schritte wird dann aus `timestep` und diesem Wert berechnet. Muss angegeben werden, wenn `number` nicht angegeben wird. Darf nicht gleichzeitig mit `number` verwendet werden.
- `body`: Gibt einen Körper an, der zur Liste der zu simulierenden Körper hinzugefügt wird. Das Format für Körper wird in Abschnitt 3.8.1 erklärt.
- `bodies_file u`: Gibt eine Textdatei an, die eine Liste an Körpern enthält entsprechend des Körperformats enthält. Jede Zeile muss genau einen Körper beinhalten. Alle Körper in der Datei werden den zu simulierenden Körpern hinzugefügt.

Die folgenden Argumente steuern die Ausgabeparameter für alle Varianten des Programms:

- `print-every n` (`p`): Alle n Schritte werden die aktuellen Positions- und Geschwindigkeitsdaten über den Standard-Ausgabestrom ausgegeben.
- `output-to path` (`o`): Gibt den Ausgabepfad an. Wenn diese Option angegeben wird, muss auch `output-format` angegeben werden. Wenn diese Option nicht gegeben ist, findet keine Dateiausgabe statt.
- `output-format [netcdf hdf5]` (`f`): Gibt an, welches Ausgabeformat verwendet werden soll. Wenn diese Option angegeben wird, muss auch `output-to` angegeben werden.
- `output-strategy [fpp sfsw sfcw sfmw]` (`s`): Gibt die Ausgabestrategie an. `fpp` und `sfsw` stehen für alle Varianten zur Verfügung. `sfcw` und `sfmw` stehen nur für die Variante `MPI` zur Verfügung. Wenn `sfcw` angegeben wird, muss auch `output-strategy-writers angegeben werden`.
- `output-strategy-writers n` (`w`): Gibt an, wie viele Prozesse für die Strategie `sfcw` an der Ausgabe teilnehmen. `n` muss zwischen 1 und der Anzahl an verfügbaren MPI-Prozessen liegen. Muss genau dann angegeben werden, wenn für `output-strategy sfcw` angegeben wird.

In der Variante `POSIX` kann außerdem `posix-threads n` (`x`) angegeben werden, um die Anzahl der berechnenden POSIX-Threads zu steuern. Es werden $n - 1$ Threads erzeugt. Zusammen mit dem ursprünglichen Thread rechnen dann n Threads.

3.8.1. Körperformat

Ein Körper wird anhand von fünf Eigenschaften definiert. Diese sind `x`, `y`, `vx`, `vy`, und `m`. `x` und `y` geben die Position an. `vx` und `vy` geben die Geschwindigkeit an. `m` gibt die Masse an.

Ein valider String, der einen Körper definiert, ist nach dem Muster `<Eigenschaft>=<Wert>[,<Eigens` aufgebaut. Die Masse muss immer angegeben werden. Wenn zwei Körper auf dem gleichen Punkt liegen, wird die Berechnung fehlerhafte Werte produzieren.

4. Implementationsdetails

In diesem Kapitel wird auf die Umsetzung der im Entwurf erläuterten Ansätze eingegangen. Dabei stehen Details der Parallelisierungsansätze sowie problemspezifische Implementationsdetails im Vordergrund.

4.1. Entwicklungsumgebung

Das Programm wurde auf einer Instanz der Linux-Distribution Debian entwickelt. Die primäre Ausführungsumgebung war das Cluster des Arbeitsbereiches Wissenschaftliches Rechnen an der Universität Hamburg. Die Abhängigkeiten wurden mit dem Paketmanager Spack installiert und verwaltet. Das Programm wurde mit dem `mpicc`-Compiler kompilliert, welcher ein Teil des OpenMP-Pakets ist und im Hintergrund den GCC-Kompiler verwendet.

4.2. Zentraler Algorithmus

Der zentrale Algorithmus wird übersichtlich gehalten, indem der berechnende Code in Funktionen ausgelagert werden. Insbesondere erleichtert das die Umsetzung der POSIX-Thread-Implementation, erläutert in 4.4

4.3. Regressionstests

Für die Ausgabe mit HDF5 und NetCDF werden die Kommandozeilenprogramme dieser Bibliotheken verwendet, um die Ausgabedateien mit den Referenzdateien zu vergleichen.

4.4. POSIX-Threads

Die Daten, die zur Ausführung eines Threads benötigt werden, werden am Anfang der Ausführung initialisiert. Für jeden Thread, der erstellt werden soll, wird ein `posix_data_t`-Objekt erzeugt und mit den notwendigen Daten befüllt. Anhand dieser Daten werden die Threads vom primären Thread aus gestartet. Bevor die Ergebnisse mit `pthread_join` gesammelt werden, führt der primäre Thread selber die Berechnungen anhand des nullten Datensatzes aus. Dank der Auslagerung der Berechnungen benötigt die Umsetzung der für `pthread_create` notwendigen Funktion wenig duplizierten Code. Dennoch ist

es nötig, die berechnenden Schleifen zu kopieren. Es wird eine Barriere erzeugt und zwischen den Schleifen ein `pthread_barrier_wait`-Aufruf eingefügt, damit die Trennung zwischen den Phasen der Simulation gewährleistet ist.

4.5. OpenMP

Es genügt drei Pragmas, um die Parallelisierung umzusetzen. Das erste wird an die die Kraft berechnende Schleife gesetzt:

```
1 #ifdef NBODY_OPENMP
2 #pragma omp parallel for
3 #endif
4 for (size_t j = from; j < to; j++) {
5     apply_forces(bodies, j, timestep_size);
6 }
```

Zeile 1 und 3 dienen dazu, diese Optimierung nur für die OpenMP-Variante zu aktivieren. Zeile 2 ist das parallelisierende pragma. Es weist an, das hier eine parallelisierbare for-Schleife vorliegt.

Das zweite Pragma wird an die die Positionen anpassende Schleife gesetzt.

```
1 #ifdef NBODY_OPENMP
2 #pragma omp parallel for reduction(+:total_distance)
3 #endif
4 for (size_t j = from; j < to; j++) {
5     vector3 this_distance = move(bodies, j, timestep_size);
6     total_distance = add(total_distance, this_distance);
7 }
8 #endif
```

Dieses pragma ist um eine Reduktion erweitert, die die berechneten „total_distance“-Werte aufsummiert. Diese Reduktion wird automatisch nach der Ausführung aller parallelen Sektionen durchgeführt und hinterlässt im nachfolgenden sequentiellen Code den errechneten Wert.

Da „total_distance“ vom Datentyp `vector3` ist muss die Reduktionsfunktion bereitgestellt werden. Das dafür notwendige dritte pragma findet sich in `vector3.h`:

```
1 vector3 add(vector3 summand1, vector3 summand2);
2 #pragma omp declare reduction(+ : vector3 : \
3         omp_out = add(omp_out, omp_in)) \
4         initializer(omp_priv = omp_orig)
```

Die zweite Zeile deklariert eine +-Reduktion über den Datentyp `vector3`. Die dritte Zeile definiert für diese, dass der Ausgabewert damit berechnet wird, die Funktion `add` aus der ersten Zeile mit dem bisherigen Ausgabewert und dem nächsten Eingabewert aufzurufen. Schließlich wird der Initialwert für diese Reduktion als gleich dem Wert angegeben, der in dem zu parallelisierenden Code als Ursprungswert angegeben ist.

Es ist möglich, mithilfe weiterer Parameter eine bessere Nutzung der verfügbaren Ressourcen zu erreichen. Eine solche Abstimmung ist zwar Sinnhaft, findet hier aber keine Anwendung. Zum einem ist der Effekt einer solchen Abstimmung Plattformabhängig, zum anderen ändert sich wenig an der Verwendung. Der Effekt und der dafür nötige Aufwand sind hinreichend illustriert.

4.6. MPI

Die Umsetzung der Prozesskommunikation benötigt zwei Aufrufe. Zuerst wird `MPI_Allgatherv` eingesetzt, um die verschieden großen Datenblöcke der einzelnen Prozesse mit allen anderen Prozessen auszutauschen.

```
1 MPI_Allgatherv(MPI_IN_PLACE, to - from, vector3_typeid,
     bodies.positions, recieve_counts, displacements,
     vector3_typeid, MPI_COMM_WORLD);
```

Die Menge an Elementen, die von diesem Prozess beigetragen werden ist durch `to - from` gegeben. Für diesen Prozess p_n ist `from` gleich $U(p_n)$ und `to` gleich $O(p_n)$ (siehe Abschnitt 3.6.1). `MPI_IN_PLACE` gibt an, dass das Array, in das die empfangenen Daten geschrieben werden, auch die Quelle für die Daten ist.

Die Arrays `recieve_counts` und `displacements` geben respektive an, wie viele Elemente von einem gegebenen Prozess empfangen werden und an welcher Stelle im Empfangsarray die Daten geschrieben werden sollen. Sie werden im initialisierenden Code vorbereitet. Dabei wird `recieve_counts` an Stelle n auf $B(n)$ und `displacements` auf $U(n)$ gesetzt.

Um die Gesamtdistanz für die Rezentrierung allen Prozessen bekannt zu machen, berechnet jeder Prozess seinen Teil der Gesamtsumme. Diese wird mit `MPI_Allgather` in einem eigens dafür angelegten Array ausgetauscht und pro Prozess aufaddiert. Da hier jeder Prozess genau einen Wert beiträgt ist die -v Variante der Funktion nicht nötig.

```
1 distances[rank] = total_distance;
2 MPI_Allgather(MPI_IN_PLACE, 1, vector3_typeid, distances,
     1, vector3_typeid, MPI_COMM_WORLD);
3 if(rank == 0) {
4     for(int i = 0; i < size; i++){
5         if(i != rank) total_distance = add(total_distance,
              distances[i]);
6     }
7 }
```

4.7. Parallele Ausgabe

Der Code zum Datenaustausch für die Ausgabestrategien zeigt die Komplexität des Codes eines komplexeren MPI-Kommunikations-Schemas:

```
1  MPI_Status status;
2  int from = calculate_from(schema.rank, schema.size,
       total_count);
3  if(schema.am_participating) {
4      int recieved_so_far = 0;
5      for(int i = 0; i < schema.size; i++){
6          int would_recieve = schema.recieves[i];
7          if(would_recieve > 0) {
8              MPI_Recv(buffer + from + recieved_so_far,
                   would_recieve, schema.datatype, i, 0,
                   MPI_COMM_WORLD, &status);
9              recieved_so_far += would_recieve;
10         }
11     }
12 }
13 if(schema.send_to != schema.rank){
14     MPI_Send(buffer + from, schema.send_size, schema.
           datatype, schema.send_to, 0, MPI_COMM_WORLD);
15 }
```

4.7.1. HDF5

Falls die MPI-Variante kompiliert wird, werden auch die notwendigen Attributslisten erzeugt, um den parallelen Zugriff durchzuführen.

Zum Schreiben des ersten Datensatzes wird die den `write_file`-Schritt implementierende Funktion aufgerufen. Diese Funktion berechnet aus dem gegebenen Rang und der Größe den zu schreibenden Datenabschnitt. Sie erweitert die Datei und schreibt den Datenabschnitt. Da beide Datensets die gleichen Eigenschaften aufweisen unterscheiden sich die verdoppelten Aufrufe kaum. Die Auswahl eines Hyperslabs und der schreibende Aufruf konnten zusammengefasst und als gemeinsame Funktion ausgelagert werden.

5. Evaluation

Die Varianten und Ausgabemodule werden auf ihre Effektivität hin analysiert. Ihre Performanz wird miteinander verglichen und ihre zugrundeliegenden Bibliotheken analysiert. Abschließend wird das Programm im Hinblick auf die Lehre aufgeschlüsselt. Es werden mögliche Übungsaufgaben sowie das für sie notwendige Vorwissen identifiziert.

5.1. Methodik

Die Parallelisierungen werden in zwei Schritten analysiert. Zuerst wird eine Performanzanalyse durchgeführt. Die Ergebnisse dieser werden mit den Erwartungen aus den theoretischen Modellen abgeglichen und eine Zusammenfassung für den Effekt der Varianten gegeben.

Anschließend werden die verwendeten Code-Bibliotheken analysiert. Ihr Einfluss auf die Codequalität wird betrachtet und es wird eine Kosten-Nutzen-Einschätzung abgegeben, die den Implementationsaufwand gegen die Ergebnisse aus der Performanzanalyse aufwiegt. Im Falle der Varianten wird auf die nötigen Modifikationen am Code des zentralen Algorithmus eingegangen. Die Ausgabe-Bibliotheken werden im Bezug auf die Umsetzung ihres Moduls betrachtet.

5.1.1. Testumgebung

Die Tests werden auf dem Supercomputer Levante des Deutschen Klimarechenzentrum ausgeführt. Levante ist ein Atos BullSequana XK2000 und besteht aus zwei Partitionen, einer GPU- und einer CPU-Partition. Für die Ausführung werden die Knoten der CPU-Partition verwendet, welche CPUs vom Typ AMD 7763 mit mindestens 256 Gigabyte Hauptspeicher besitzen. Jeder Knoten hat 64 Kerne zur Verfügung und kann bis zu 128 Threads parallel ausführen. Diese Knoten sind mit HDR-InfiniBand vernetzt, welches eine Datenübertragungsleistung von bis zu 100 Gigabit pro Sekunde erreicht. Insgesamt erreicht Levante eine Spitzenleistung von 14 PetaFLOPs. [3]

Levante wird mit dem Betriebssystem Red Hat Enterprise Linux [26] betrieben. Software wird mit dem `modules`-System verwaltet. [4] Es wird `slurm` für die Job- und Knotenverwaltung verwendet. Es ist ein Zeitlimit von acht Stunden für diese Nodes konfiguriert, d.h. ein individueller Job wird nach acht Stunden Laufzeit abgebrochen.

Levante verwendet ein Festplattensystem von DDN mit insgesamt 130 Petabyte. Es ist in mehrere Dateisysteme aufgeteilt, *HOME*, *WORK*, und *SCRATCH*. Alle Dateisysteme verwenden Lustre als Dateisystem. *SCRATCH* wird verwendet, um die Ausgabedaten zu speichern. Auf diesem stehen fünfzehn Petabyte zur Verfügung. Diese sind so kon-

figuriert, dass für Datien unter einem Gigabyte ein Streifen, zwischen einem und vier Gigabyte vier Streifen und ab vier Gigabyte 16 Streifen verwendet werden. [3]

5.1.2. Tests

Es wird ein einfaches, auf das Programm zugeschnittenes Testframework in Python implementiert, um die Performanz zu messen. Die auszuführenden Tests werden im Code des Frameworks definiert. Sie sind in Testreihen aufgeteilt, die jeweils einen auf einen bestimmten zu testenden Aspekt abzielen. Die Testreihen sind zu sogenannten *Suites* zusammengefasst, die ein ähnliches Thema haben und zusammen innerhalb des Zeitlimits der Testumgebung ausgeführt werden können. Die auszuführende Suite wird mit einem Kommandozeilenargument ausgewählt.

Innerhalb einer Suite können verschiedene Varianten des Programms getestet werden. Das Framework kompiliert die zu testende Variante automatisch. Alle Tests, die sich auf jeweils eine Variante beziehen, werden gesammelt und gemeinsam ausgeführt, um unnötige Rekompilation zu vermeiden. Jeder individuelle Test wird fünf mal ausgeführt. Die Messdaten werden in einem verschachtelten `dictionary` gespeichert, der die Messdaten der Variante und den Parametern zuordnet. Außerdem wird eine Übersicht der Ergebnisse erstellt und ausgegeben.

Die Ausführung ist auf die Testumgebung zugeschnitten. Mithilfe eines entsprechen konfigurierten Slurm-Skriptes werden zwei Knoten allokiert. Das Task-Maximum wird auf 128 gesetzt. Es werden die in der Dokumentation von Levante vorgeschlagenen Einstellungen verwendet.

Für die Variante MPI wird `nbody` nicht direkt ausgeführt. Stattdessen wird der Slurm-Befehl `srun` mit dem Kommandozeilenargument `-n` verwendet, um das Programm auszuführen. Dieser Befehl übernimmt dann die Verwaltung der MPI-Umgebung und erschafft so viele Prozesse, wie mit `-n` angegeben wurden.

5.1.3. Vampir

Für die Analyse der Parallelisierung mit MPI wird Vampir eingesetzt. Mit Vampir lässt sich die Ausführung von MPI-Programmen auf ihre Kommunikation hin analysieren. [31] Mithilfe des Programms scorep wird der Code so kompilliert, das während der Ausführung Spurdaten gesammelt werden, die anschließend mit Vampir eingelesen und dort analysiert werden können.

5.2. Eine Notiz zu den Messwerten

Die Messwerte verhalten sich äußert stabil. Die Varianz zwischen verschiedenen Läufen bewegt sich in den meisten Fällen bei unter einer Sekunde, selbst wenn die Ausführung länger als 2000 Sekunden dauert. Ein Beispiel dafür ist die sequentielle Ausführung mit 1.000 Körpern und 5.000 Schritten, deren schnellster Lauf 1042,72 und langsamster Lauf 1043,07 Sekunden benötigt. Es gibt nur wenige Ausnahmen davon:

- Ausführungen, deren Durchschnittliche Zeit unter einer Zehntelsekunde liegt. Für diese ist Messungenauigkeit eine hinreichende Erklärung. Die Analyse dieser Zeiten liefert keinen Erkenntnisgewinn.
- Ausführungen der Variante `OPENMP`. Da die Anzahl der Prozesse bei OpenMP dynamisch gewählt wird, sind die Schwankungen vermutlich darauf zurückzuführen.

Aufgrund dieser Stabilität der Ergebnisse wird auf eine statistische Analyse der Ergebnisse verzichtet. Schwankungen unter einem Prozent des Gesamtmesswertes werden als nicht für die Validität der Ergebnisse und Schlussfolgerungen relevant bewertet.

5.3. Performanz der Varianten

Im ersten Schritt wird die Performanz des sequentiellen Grundprogramms entlang der zu berechnenden Schritte und der Anzahl der Körper analysiert. Erwartet wird, dass gemäß der Komplexität $O(n^2)$ die Dauer der Berechnung quadratisch mit der Zahl der Körper skaliert. Gleichzeitig wird erwartet, das die Dauer linear mit der Menge an Zeitschritten steigt. Um dieses Verhalten zu testen, wird die sequentielle Variante mit mehreren Schrittzahlen zwischen 100 und 10.000 sowie Körperzahlen zwischen 10 und 500 ausgeführt. Eine Ausgabe findet dabei nicht statt. Die Ergebnisse sind in Grafik 5.2 visualisiert. Es zeigt sich, dass die Ausführungszeit entsprechend den Erwartungen skaliert. Die Kombination aus 10 Körpern mit 10.000 Schritten dauert etwa genauso lange zu berechnen wie die Kombination aus 100 Körpern und 100 Schritten. Beide benötigen circa 0,25 Sekunden. Wenn für 100 Körper die Schrittzahl von 100 auf 1.000 erhöht wird, steigt auch die Ausführungszeit von etwa 0,25 Sekunden auf etwa 2,1 Sekunden. Diese leichte Abweichung von der erwarteten Skalierung auf 2,5 Sekunden lässt sich darauf zurückführen, dass die Messungenauigkeit zwischen 0,05 und 0,1 Sekunden liegt. Eine Steigerung auf 1.000 Körper wiederum führt zu einem Anstieg von etwa 2,1 Sekunden auf etwa 211 Sekunden.

Im zweiten Schritt wird festgestellt, ob die parallelen Varianten einen Speedup erzielen. Die Varianten `POSIX` und `MPI` werden mit acht Prozessen ausgeführt. Der Algorithmus ist vollständig parallelisierbar. Außerhalb von Fixkosten in der Implementation ist der theoretisch erreichbare Speedup also gleich der Prozessorzahl. Damit wird in diesem Durchlauf mit einem Speedup von 8 für die Varianten `POSIX` und `MPI` gerechnet.

`OPENMP` ist ein Sonderfall. Die Anzahl der verwendeten Threads lässt sich zwar theoretisch direkt kontrollieren, die Standard-Konfiguration aber wählt die Anzahl der Threads automatisch anhand der Last und der verfügbaren Kerne. Da die Knoten 128 Prozessoren besitzen hängt der erwartete Speedup damit von dieser Automatik ab, sollte aber nie mehr als 128 betragen.

Ausgeführt wird einmal mit 100 Körpern und 10.000 Schritten, einmal mit 1.000 Körpern und 10.000 Schritten. Es ergeben sich die Ausführungszeiten in Tabelle 5.1.

Der beobachtete Speedup ist für 100 Körper deutlich geringer als erhofft. Eine Steigerung der Körperzahl nähert den Speedup an den erwarteten Wert an. Diese Beobachtung lässt sich damit erklären, dass die Fixkosten der Ansätze zwischen den Iterationen des

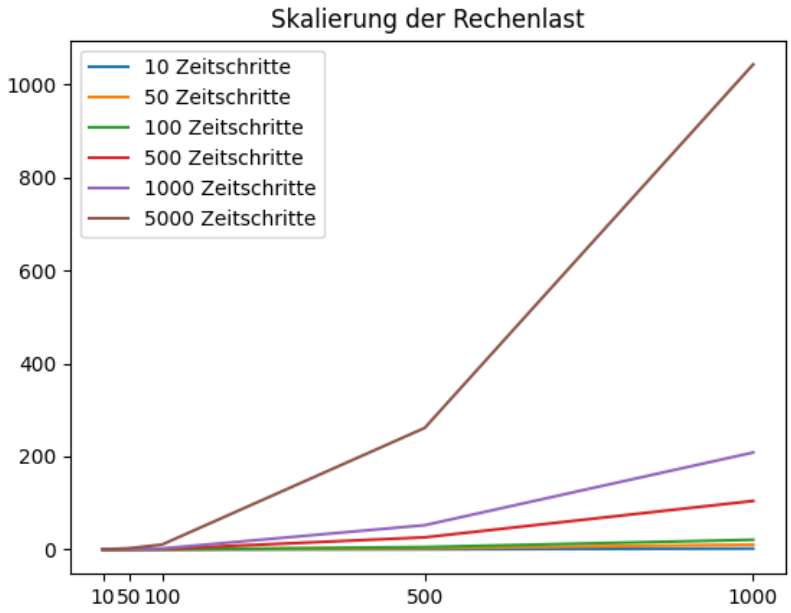

Abbildung 5.1.: Ergebnisse der Messungen der sequentiellen Variante.

Körper	Schritte	`SEQUENTIAL` t	`POSIX` t_p (Speedup)	`OPENMP` t_p (Speedup)	`MPI` t_p (Speedup)
100	10,000	21,17s	5,39s (3,93)	8,96s (2,36)	4,09s (5,18)
1000	10,000	2085,40s	266,71s (7,82)	20,98s (99,40)	267,09s (7,81)

Tabelle 5.1.: Ausführungszeiten und Speedup der Varianten mit 8 Prozessen.

Algorithmus stattfindet. Diese Beobachtung wird mit weiteren Testläufen mit verschiedenen Körperzahlen und Prozesszahlen bestätigt.

Es ist ersichtlich, dass alle Varianten Parallelisierung nahe dem theoretischen Limit ermöglichen.

Die Gebundenheit des Programms wird überprüft, indem der vorherige Testlauf mit aktivierter Ausgabe wiederholt wird. Die Ergebnisse zeigen, dass das Programm zumindest im Umfeld der gewählten Parameter CPU-gebunden ist. Es wird geprüft, ob die Ausgabe der Kräfte zu einer Ausgabe-Gebundenheit führt. Dafür wird das Programm mit aktivierter und deaktivierter Ausgabe der Kräfte verglichen. Es wird überprüft, ob eine größere Prozesszahl im ersten Fall zu einem ähnlichen Speedup führt. Die Abbildung 5.3 zeigt, dass ab einer bestimmten Prozesszahl kein weiterer Speedup erreicht wird. Das zeigt, dass in diesem Fall das Programm nicht CPU-gebunden ist.

Um zusätzlich zu zeigen, dass das Programm tatsächlich Ausgabe-gebunden ist, wird

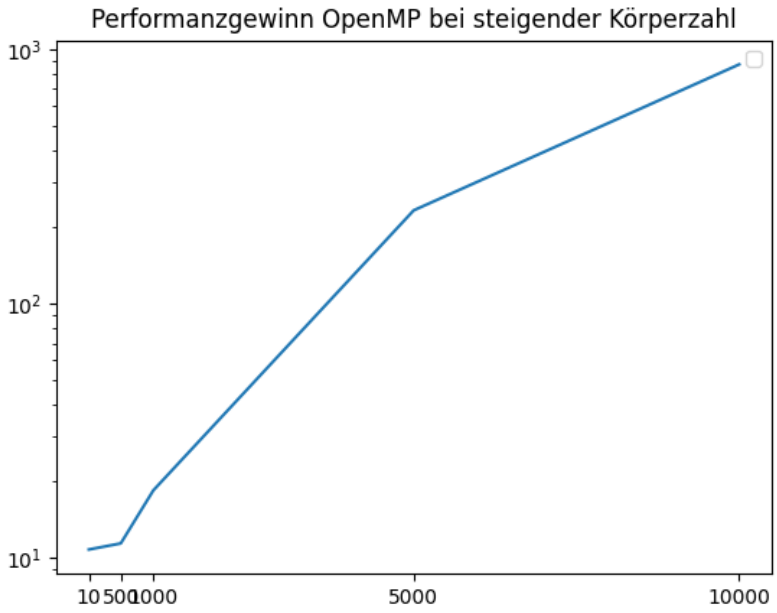

Abbildung 5.2.: Skalierung der Ausführungszeit von OpenMP mit der Anzahl der Körper. Es ist klar erkennbar, dass die Ausführungzeit weniger als quadratisch skaliert.

die Parallelisierung der Ausgabe angewendet. Wenn diese den Speedup verbessert, wo weitere Prozesse das nicht erreicht haben, dann ist das Programm Ausgabe-gebunden. Dafür werden die Ausgabestrategien SFCW und FPP konfiguriert. Dieser Test konnte leider nicht erfolgreich vor Ablauf der Frist ausgeführt werden. In den vorbereitenden Ausführungen wurde festgestellt, dass die Kombination aus dem Ausgabeformat NetCDF und der Strategie SFCW mit zwei Schreibern die Ausführungszeit ungefähr halbiert wurde. Damit ist davon auszugehen, dass eine Ausgabegebundenheit vorliegt.

5.4. Analyse der Ansätze

5.4.1. POSIX-Threads

Die Umsetzung von POSIX-Threads erforderte kleinere Anpassungen an der Architektur des Programms. Die Daten für diese müssen recht kleinteilig verwaltet werden. Neue Ein- und Ausgabedaten hinzuzufügen erfordert Änderungen an mehreren Stellen. Zusätzlich wird dieser Ansatz dadurch untergraben, dass OpenMP ein effektiverer Weg ist, das gleiche Programmverhalten umzusetzen. Unter anderem deshalb werden POSIX-Threads in wissenschaftlicher Software selten direkt eingesetzt, im Gegensatz zu OpenMP. Ihre Performanz ist dennoch vergleichbar mit den anderen Ansätzen und stehen diesen in

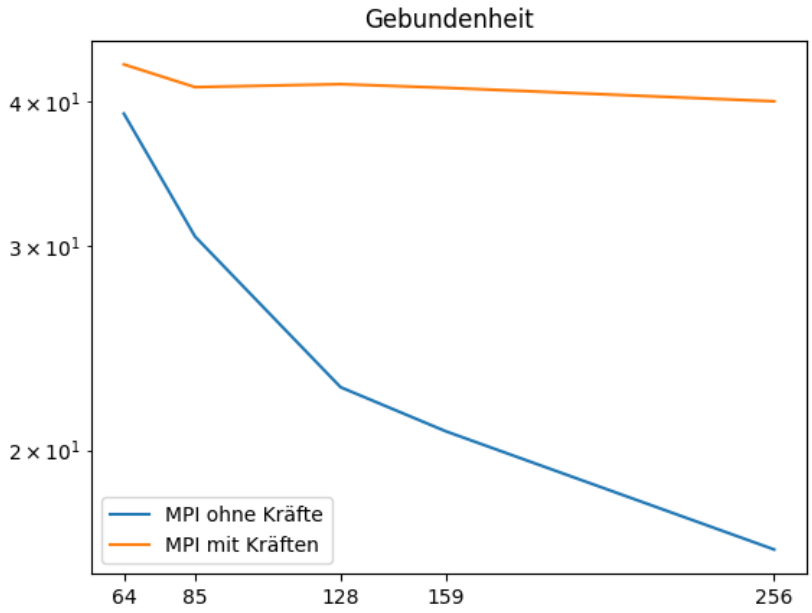

Abbildung 5.3.: Ausführungszeit der Variante `MPI` mit und ohne aktivierter Ausgabe der Kräfte. Es ist eindeutig, dass ohne diese eine CPU-Gebundenheit vorliegt.

rohem Speedup in nichts nach.

Sie dienen als gutes Schaubeispiel für die zugrundeliegenden Konzepte für Threading und Parallelismus in MIMD-Maschinen. Durch die Prävalenz von Threads auch außerhalb des Themenfeldes der Parallelisierung sind sie außerdem sehr vielseitig. Sie lassen sich einfach an andere Problembereiche anpassen und ihre Anwendung ist sehr ähnlich zu den Threading-Konzepten anderer Sprachen, wie z.B. Rust oder Python. Die durch sie vermittelten Konzepte stellen ein nützliches Werkzeug im Repertoire eines Programmierers dar.

5.4.2. OpenMP

Die Umsetzung des OpenMP-Ansatzes erforderte kaum Aufwand. Gleichzeitig zeigt sich ein Speedup von 99, sehr nahe dem theoretischen Maximum. Dieses Kosten-Nutzen-Verhältnis ist das Beste der analysierten Ansätze. Dafür bestehen auch die restriktivsten Voraussetzungen, um so angewendet werden zu können. Zwar gibt es viele weitere Konzepte in OpenMP, die auch andere Anwendungszwecke erlauben, aber diese erfordern eine weitere Einarbeitung, die weit über die hier gezeigte Parallelisierung hinausgehen.

Insgesamt zeigt sich OpenMP als effektiver erster Schritt einer Parallelisierung. Wenn das gegebene Problem mit OpenMP parallelisierbar ist, dann wird mit wenig Aufwand

viel erreicht. Anschließend kann durch weitere Analyse und ein tieferes Verständnis der Bibliothek prinzipiell noch mehr mit OpenMP erreicht werden.

5.4.3. MPI

MPI erfordert Bedenken über die gesamte Struktur des Codes. Da das gesamte Programm mehrmals gestartet wird muss potentiell die Verwendung von externen Ressourcen koordiniert werden. Änderungen am Code müssen diese mehrfache Ausführung beachten.

Wenn die Grundstruktur des Codes entsprechend entworfen wurde, erfordert die Umsetzung wenige Eingriffe. Der notwendige initialisierende Code ist simpel und benötigt selten Anpassungen. Gleiches gilt für den abschließenden Code.

Die Komplexität der Umsetzung skaliert mit der Komplexität des Kommunikationsschemas. Das für den zentralen Algorithmus notwendige Schema ist einfach, entsprechend übersichtlich gestaltet sich die Umsetzung. Diese Koordinierung der Prozesse benötigt das Schema nach 3.7.2, um zu funktionieren, und zeigt einen deutlich komplexeren ausführenden Code.

Gleichzeitig zeigt dieser Code die Flexibilität von MPI. Anhand einer einfachen Datenstruktur mit ein paar Garantien über die enthaltenen Daten können verschiedene komplexe Strategien mit dem gleichen Code ausgeführt werden.

5.4.4. IO

NetCDF

Die Umsetzung von NetCDF stellt sich als unkompliziert heraus, auf Kosten der Freiheiten, die HDF5 ermöglicht. Eine simple Ausgabe zu implementieren erfordert wenig Vorwissen. Von da an kann die Ausgabe Stück für Stück verbessert und ausgebaut werden. Die Schnittstelle zeichnet sich durch einen geringen Verwaltungsaufwand aus. Es gibt wenige zu beachtende Nebenwirkungen und wenige zu verwaltende Variablen. Gleichzeitig ist die Performanz von NetCDF zumeist besser als die von HDF5 bei einer ähnlich aufgebauten Ausgabe.

HDF5

HDF5 stellt sich als sehr mächtig, und gleichzeitig sehr schwer zu handhaben heraus.

Die Umsetzung der HDF5-Ausgabe benötigt die Handhabung vieler Konzepte. So müssen zum Beispiel Aus den definierten Datensets zwei verschiedene, subtil unterschiedliche Datenräume erzeugt werden, die den gleichen Begriff verwenden, aber unterschiedliche Konzepte des Schreibprozesses abbilden. Zusätzlich haben diese Objekte jeweils einen Zustand, für dessen Konfiguration sich keine opportune Stelle im Code ergibt. Der Code wird durch eine weite Verteilung sich beeinflussender Anweisungen schwer nachvollziehbar. Zusätzlich sind einige der Fehlermeldungen, die die Bibliothek bei falscher Verwendung erzeugt, schwer nachzuvollziehen.

Letzterer Umstand wird dadurch erschwert, dass die Schnittstelle teilweise schwer für Menschen lesbar ist. Die Funktionen sind so benannt, dass auf den String „H5“ ein oder mehrere Großbuchstaben folgen, die das Interface angeben, gefolgt von einem die Funktion beschreibenden Namen. Beispiele sind `H5Pcreate` und `H5Dcreate`. Solche nah beieinander stehenden und ähnlich aussehenden Zeichen sind schwerer zu unterscheiden [27], was zur Verwendung der falschen Funktion führen kann.

Im Gegenzug für diese Schwierigkeiten der Umsetzung erhält man sehr mächtige Werkzeuge, um die Anwendung auf Performanz zu trimmen. Durch die Vielzahl an Konfigurationsmöglichkeiten und die genaue Umsetzung der für das Schreiben relevanten Konzepte lassen sich die Schreibvorgänge theoretisch genau an die gegebenen Umstände anpassen. Das Datenformat erlaubt es dabei, auch komplexe Zusammenhänge darzustellen.

5.5. Das Programm als Lehrgegenstand

Durch die breite Abdeckung von Konzepten eignet sich das Programm als Grundlage für Übungsaufgaben im Feld des Hochleistungsrechnens. Insbesondere kann es eine gemeinsame Grundlage für Studenten bieten. Es kann sowohl der Ausgangspunkt sein, von dem Studenten aus arbeiten, als auch als Vergleichsobjekt für die angefertigten Arbeiten.

Grundsätzlich müssen die Studierenden dafür eine gewisse Expertise mit C aufweisen. Sie müssen in der Lage sein, fremden Quellcode zu lesen und zu verstehen, wobei diese Fähigkeit auch in der Arbeit an der Software geübt wird. Außerdem müssen sie den Umgang mit Spack, slurm und Linux beherrschen, um die Software kompilieren und ausführen zu können.

5.5.1. Das Programm als Projektziel

Ein möglicher Ansatz ist, die Studierenden das Programm Schritt für Schritt implementieren zu lassen, und nach jedem Schritt eine reduzierte Variante des Programms als neuen Ausgangspunkt zu geben. Zuerst wäre die Aufgabe, eine einfachste N-Körper-Simulation zu schreiben, die der Schnittstellendefinition aus Abschnitt 3.8 entspricht.

Diese Implementationen können in der Gruppe besprochen und mit der Referenz verglichen werden. Als Aufgabe kann gestellt werden, das Skalierungsverhalten des Problems zu identifizieren. Besonderes Augenmerk kann dabei auf die Qualitätsmerkmale gelegt werden, da diese im Gegensatz zur technischen Spezifikation sehr viel Raum für Erfahrungswerte bietet.

Im Laufe des Projekts können dann die einzelnen Parallelisierungen nacheinander implementiert werden. Für die Reihenfolge ergäbe es Sinn, bei den POSIX-Threads anzufangen, um das grundlegende Konzept von parallel laufendem Code zu erklären. OpenMP kann dann genutzt werden, um aufzuzeigen, dass das manuelle erzeugen von Threads sehr kleinteilig ist und es bessere Alternativen gibt. Um tiefer in OpenMP einzusteigen kann als Aufgabe gestellt werden, die Umsetzung mit zusätzlichen Parametern zu optimieren. Schließlich kann zu MPI übergegangen werden. MPI erfordert eine klare Abgrenzung in der Vorbereitung, um den Unterschied zwischen Threads und Prozessen

und die Konsequenzen daraus zu verdeutlichen.

Es kann jedes mal ein Speedup als Ziel für die Studierenden definiert werden, damit sie den Effekt ihrer Arbeit sowohl sicherstellen als auch gezielt wahrnehmen, um einen spürbaren Lernerfolg zu erzielen.

Mit dem Wissen über MPI kann auf die parallele Ausgabe eingegangen werden. Da HDF5 und NetCDF sehr ähnlich zueinander funktionieren, aber HDF5 deutlich mehr Konzepte erfordert, ist NetCDF die sinnvollere Wahl als Einstieg in das Thema. Ein erstes Ziel wäre es, die bisherige Kommandozeilenausgabe durch eine Dateiausgabe zu ersetzen. Im Zuge dessen kann auch die Ausgabe der Kräfte aufgegriffen werden, um CPU- vs. Ausgabe-Gebundenheit zu illustrieren.

Anschließend können die Ansätze paralleler Ausgabe erläutert werden. Für einen tieferen Einstieg in MPI kann hier der Ausgabeschema-Algorithmus verwendet werden, entweder indem der Algorithmus vorgegeben und eine Implementation gefordert wird, oder indem die Studierenden auch selber einen Algorithmus entwickeln sollen. Alternativ kann ein bestimmtes Ausgabeschema vorgegeben werden. Unabhängig vom gewählten Schema können die Studierenden dann die parallele Ausgabe von NetCDF aktivieren und ausprobieren, ob sie einen Speedup feststellen können. Dieser Schritt erfordert allerdings Zugriff auf ein paralleles Dateisystem. Wenn kein solches vorhanden ist, ist kein Erfolgserlebnis zu erwarten. Der Prozess kann gleichermaßen mit HDF5 durchgeführt werden.

In jedem Schritt kann es dabei freigestellt sein, die Referenzimplementation zu übernehmen oder die eigene weiter zu verwenden. Wenn Studierende sich dafür entscheiden, den eigenen Entwurf weiter zu verfolgen, ist das eine Gelegenheit, in der Gruppe Entwurfsentscheidungen kritisch und differenziert zu betrachten. Das gibt die Möglichkeit, praktische Erfahrungen im Bezug auf Softwarequalität und ihre Auswirkung auf den Entwicklungsprozess zu sammeln.

5.5.2. Freie Aufgaben am Programm

Der Code kann auch direkt als Grundlage für Aufgaben dienen. Jeder der Teilschritte in dem Projektansatz kann auch als eigenständige Aufgabe gestellt werden, indem die entsprechenden Teile des Codes entfernt werden und der modifizierte Code an die Studierenden gegeben wird.

Außerdem kann der Algorithmus als Optimierungsobjekt verwendet werden. Es gibt eine Reihe an möglichen Optimierungen, wie die in Abschnitt 3.4 erwähnte Halbierung der Kräftematrix. Es kann als Aufgabe gestellt werden, eigenständig nach solchen Optimierung zu suchen und sie zu implementieren, mit dem generellen Ziel einen möglichst großen Speedup zu erreichen.

Dank der recht granularen Skalierbarkeit der Ausführungszeit können auch einfache Messaufgaben am Programm durchgeführt werden. Die Studierenden können am konkreten Beispiel quadratische und lineare Skalierung ausprobieren und die Limits von Ausgabe und Skalierung erfahren.

6. Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, die praktischen und theoretischen Ansätze der Parallelisierung an einer konkreten Umsetzung zu veranschaulichen. Das N-Körper-Problem wurde als geeigneter Kandidat dafür identifiziert, da es gleichzeitig simpel genug ist, um der Veranschaulichung nicht im Weg zu stehen. Es ist parallelisierbar genug und liefert genug Ausgabedaten, um eine große Bandbreite an Konzepten behandeln zu können. Der umgesetzte Algorithmus eignet sich durch eine einfache Struktur dazu, Stärken und Schwächen der verschiedenen Ansätze hervorzuheben. Die Performanz-Charakteristiken des Programms illustrieren Speedup, das Amdahlsche Gesetz und Gebundenheit.

Durch die optionale Ausgabe-Gebundenheit ist die Effektivität von paralleler Ausgabe sichtbar. Die Gegenüberstellung von HDF5 und NetCDF zeigt NetCDF als das effektivere Werkzeug, um beispielhafte parallele Ausgabe umzusetzen.

Das Programm eignet sich als Grundlage für Übungsaufgaben, insbesondere als Referenzarbeit für eine gestaffelte Projektarbeit und als Grundlage für freie Optimierungsaufgaben.

Die breite der Arbeit führte dazu, dass die einzelnen Aspekte nur recht oberflächlich beleuchtet werden konnten und ihre Umsetzung nur einen optimalen Fall illustrieren. OpenMP-Tasks sind ein aktuelles Thema, welches keine Anwendung fand und in weitere Arbeit für die Lehre aufgegriffen werden kann. Außerdem ist die praktische Umsetzung von komplexeren Algorithmen auch ein für Studierende betrachtenswertes Thema, das weiter ausgeführt werden könnte. Dafür ist entweder die Umsetzung des gleichen Problems mit einem komplexeren Algorithmus oder aber die Wahl eines anderen Problems nötig.

A. Kompilierung und Abhängigkeiten

Es wird ein Build-Skript bereitgestellt, `build.sh`. Es verwendet `make` und `spack`. Das dabei aufgerufene `Makefile` verwendet `mpicc`. Für diese Arbeit wurde das Programm mit Version 3.4.2 kompiliert, welche wiederum `gcc` Version 11.1.0 verwendete.

Per `spack` müssen die Pakete `netcdf-c`, `hdf5` und entweder `mpich` oder `openmp` bereitgestellt werden. Außerdem muss OpenMP vorhanden sein.

Das Build-Skript benötigt als erstes Argument die zu bauende Variante. Es muss eine aus `NBODY_SEQUENTIAL`, `NBODY_POSIX`, `NBODY_OPENMP` oder `NBODY_MPI` angegeben werden. Als zweites Argument kann `WITH_FORCES` angegeben werden, um die Ausgabe der Kräfte zu aktivieren. Ausgabe ist die ausführbare Datei `nbody`.

B. Vollständige Messdaten

Die Messdaten sind in den beiliegenden digitalen Daten als `pickle`-Dateien hinterlegt, die mithilfe der `pickle`-Bibliothek in Python eingelesen werden können, wenn das `nbody_test_base`-Modul importiert wurde. Die Schlüssel der ersten Ebene geben an, zu welcher Variante die Messdaten gehören, und ob das Abspeichern der Kräfte aktiviert wurde. In der zweiten Ebene werden die Parameter des Programms mit angegeben. Der erste Wert ist dabei immer die Anzahl der Körper und der zweite die Anzahl der Zeitschritte. Die weiteren Werte sind Paare aus einem Parameternamen und einem Wert für diesen Parameter. Die schlussendlichen Werte sind die gemessenen Ausführungszeiten in Sekunden als Liste, in der jeder Wert einer Ausführung entspricht.

Literatur

[1] Lyndon Clarke, Ian Glendinning und Rolf Hempel. „The MPI message passing interface standard“. In: *Programming environments for massively parallel distributed systems*. Springer, 1994, S. 213–218.

[2] James W Cooley und John W Tukey. „An algorithm for the machine calculation of complex Fourier series“. In: *Mathematics of computation* 19.90 (1965), S. 297–301.

[3] DKRZ. *HLRE-4 Levante*. 2022. URL: `https://www.dkrz.de/de/systeme/hpc/hlre-4-levante/`.

[4] DKRZ. *Introduction to Levante*. 2022. URL: `https://docs.dkrz.de/doc/levante/index.html`.

[5] Fvillanustre. *Thor Cluster.png*. File: `Fig2 Thor Cluster.jpg`. 2010. URL: `https://commons.wikimedia.org/wiki/File:Fig2_Thor_Cluster.jpg`.

[6] Yuepu Guo u. a. „Tracking technical debt—An exploratory case study“. In: *2011 27th IEEE international conference on software maintenance (ICSM)*. IEEE. 2011, S. 528–531.

[7] *HDF5 Dokumentation - Page Buffering*. 2022. URL: `https://portal.hdfgroup.org/display/HDF5/Page+Buffering`.

[8] Vishakh Hegde und Sheema Usmani. „Parallel and distributed deep learning“. In: *May* 31 (2016), S. 1–8.

[9] Zhong Heping u. a. „Accelerating range Doppler imaging algorithm for multiple-receiver synthetic aperture sonar on multi-core-based architectures“. In: *Soft Computing* 24.13 (2020), S. 9777–9788.

[10] Dean Hildebrand und Frank Schmuck. „Gpfs“. In: *High Performance Parallel I/O*. Chapman und Hall/CRC, 2014, S. 149–160.

[11] Mark Ilg, Jonathan Rogers und Mark Costello. „Projectile monte-carlo trajectory analysis using a graphics processing unit“. In: *AIAA atmospheric flight mechanics conference*. 2011, S. 6266.

[12] R Intel. „Intel 64 and ia-32 architectures optimization reference manual“. In: *Intel Corporation, Sept* (2022).

[13] Frank Klemm. *Visualisierung von Amdahl's Gesetz*. File: `Amdahl.jpg`. 2017. URL: `https://commons.wikimedia.org/wiki/File:Amdahl.png`.

[14] *lustre*. 2022. URL: `https://www.lustre.org/`.

[15] Sandra Mendez u. a. *PRACE Best Practice Guide - Parallel I/O*. Techn. Ber. PRACE aisbl, 2019.

[16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. Juni 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[17] *MPICH*. 2022. URL: https://www.mpich.org/.

[18] *NetCDF Users Guide*. 2022. URL: https://docs.unidata.ucar.edu/nug/current/index.html.

[19] *Network Common Data Form (NetCDF)*. 2022. URL: https://www.unidata.ucar.edu/software/netcdf/.

[20] *Open MPI: Open Source High Performance Computing*. 2022. URL: https://www.open-mpi.org/.

[21] *OpenMP*. 2022. URL: https://www.openmp.org/.

[22] *Partnership for Advanced Computing in Europe*. 2022. URL: https://prace-ri.eu/.

[23] Nils Preuss u. a. „Methods and technologies for research-and metadata management in collaborative experimental research". In: *Applied Mechanics and Materials*. Bd. 885. Trans Tech Publ. 2018, S. 170–183.

[24] *pthreads(7) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man7/pthreads.7.html.

[25] *pthreads(7) — Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man7/pthreads.7.html.

[26] Inc. Red Hat. *Red Hat Enterprise Linux*. 2022. URL: https://www.redhat.com/de/technologies/linux-platforms/enterprise-linux/.

[27] James E Sheedy u. a. „Text legibility and the letter superiority effect". In: *Human factors* 47.4 (2005), S. 797–815.

[28] Erich Strohmaier u. a. *TOP500 June 2022*. 2022. URL: https://top500.org/lists/top500/2022/06/.

[29] *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality measurement framework*. Standard. Geneva, CH: International Organization for Standardization, März 2000.

[30] *The HDF5® Library & File Format*. 2022. URL: https://www.hdfgroup.org/solutions/hdf5.

[31] *Vampir*. 2022. URL: https://vampir.eu/.

[32] Tobias Weinzierl. *Principles of Parallel Scientific Computing: A First Guide to Numerical Concepts and Programming Methods*. Springer Nature, 2021.

[33] Rosmarie Irma Wirth. „Über die Anwendung neuronaler Netze zur Messung der Top Quark Masse am LHC". In: (2018).

[34] Yoo u. a. „Slurm: Simple linux utility for resource management". In: *Workshop on job scheduling strategies for parallel processing*. Springer. 2003, S. 44–60.

[35] Junchao Zhang, Babak Behzad und Marc Snir. „Optimizing the Barnes-Hut algorithm in UPC“. In: *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2011, S. 1–11.

Eidesstattliche Erklärung

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

Hamburg, den 26.09.2022 R. Pranke

Rasmus Pranke

Veröffentlichung

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

Hamburg, den 26.09.2022 R. Pranke

Rasmus Pranke