FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

autoBLAS in verteilten Umgebungen

Masterarbeit
zur Erlangung des akademischen Grades
Master of Science
im Studiengang Informatik

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik

eingereicht von	Mark Umnus
geboren am	03.03.1998 in Bad Saarow-Pieskow

Gutachter:	Prof. Dr. Joachim Giesen
Betreuer:	Dr. Lars Kühne

Kahla, den 8. September 2021

Zusammenfassung

Massiv parallele Systeme sind das Rückgrat des modernen Hochleistungsrechnens und werden es auch im kommenden Exascale-Zeitalter bleiben. Sie bestehen aus einer Vielzahl an Knoten mit CPUs und GPUs, mitunter auch TPUs, FPGAs und ASICs, die durch Hochgeschwindigkeitsnetzwerke miteinander verbunden sind. Um die bestmögliche Performance bezüglich Geschwindigkeit, Latenz, Speicherbedarf und Energiekonsum mit diesen komplexen Maschinen zu erreichen, wurden in den letzten Jahrzehnten zahlreiche Programmiersprachen und Softwarebibliotheken entwickelt. Die vorliegende Arbeit bietet einen Überblick über eine Teilmenge dieser Lösungen und bewertet sie hinsichtlich ihrer Tauglichkeit für den Einsatz im aktuellen Umfeld. Eine Lösung zur Generierung von effizientem Code für einen einzelnen Knoten ist das Programm autoBLAS. In dieser Arbeit wird demonstriert, wie eine Erweiterung von autoBLAS zur automatischen Verteilung von Berechnungen auf mehrere Knoten prinzipiell mittels MPI und Dask realisiert werden kann. Dabei auftretende praktische Probleme werden thematisiert und Lösungsvorschläge geboten.

Inhaltsverzeichnis

1 Einführung

Obwohl die meisten Supercomputer zum Schreibzeitpunkt dieser Arbeit in der unteren Hälfte des Petaflops-Bereichs liegen, befindet sich die Welt an der Schwelle zum Exascale-Computing [83]. Damit gemeint sind Computersysteme, deren Rechenleistung die Grenze von einer Trillion *floating point operations per second* überschreitet. Das voraussichtlich erste System, das diese Grenze überschreiten wird, ist der Frontier-Supercomputer in Tennessee für 600 Millionen US-Dollar, der eine theoretische Maximalperformance von 1,5 Eflop/s mit einem GPU-CPU-Verhältnis von 4:1 erreicht [42, 85]. Enorme Rechenleistungen wie diese werden beispielsweise für Simulationen in der Medizin, Physik, Chemie oder Meteorologie eingesetzt. Die Ausschöpfung des vollen Potenzials ist jedoch nicht trivial, denn mit steigender Anzahl der Knoten im Cluster steigt tendenziell auch der Kommunikations- und Synchronisationsaufwand, wodurch Wartezeiten entstehen, in denen die Knoten keine sinnvollen Berechnungen durchführen. Um die Dauer dieser Wartezeiten zu minimieren, sind durchdachte Konzepte dafür notwendig, wie das zu lösende Rechenproblem in Teilprobleme gegliedert werden kann, die möglichst unabhängig voneinander lösbar sind. Den dafür erforderlichen Code in jedem Projekt selbst zu schreiben, ist fehleranfällig und nimmt viel Zeit in Anspruch. Aus diesem Grund gab es frühzeitig Bestrebungen, diesen Schritt durch neue Programmiersprachen und unterstützende Laufzeitsysteme zu vereinfachen. Beispiele dafür sind etwa die Programmiersprachen Cilk [12] und Split-C [27]. Optimale Lösungen wurden jedoch bisher nicht gefunden und so werden auch heute noch regelmäßig neue Programmiersprachen wie etwa Julia [11] eingeführt, die ihren Fokus auf numerisches und Hochleistungsrechnen legen. Gleichzeitig sind verhältnismäßig manuelle Implementierungsmethoden wie Fortran, C und C++ mit MPI sehr präsent, etwa im LINPACK-Benchmark für HPC-Systeme [32]. Insgesamt sind fundierte Programmierkenntnisse nach wie vor notwendig, um Zugang zum Hochleistungsrechnen zu erhalten.

Das von Lars Kühne geschaffene Programm autoBLAS versucht, diesen Umstand zu ändern. Dazu bietet es Nutzern eine Eingabesprache, die nah an der mathematischen Formulierung von Problemen und damit für viele Wissenschaftler leicht zu verstehen ist. Die Aufgabe, basierend darauf effizienten Code zu schreiben, nimmt autoBLAS seinen Nutzern ab. Bisher kann autoBLAS Code für C++, Python und SQL generieren, jedoch per se nur für einen Knoten. Dadurch bleibt den Nutzern von autoBLAS der Zugang zu massiv parallelen Systemen bisher verwehrt.

1.1 Aufgabenstellung

In dieser Arbeit soll untersucht werden, ob beziehungsweise zu welchem Grad sich autoBLAS um Fähigkeiten des verteilten Rechnens erweitern lässt. Dabei steht die Verteilung auf die Knoten von Hochleistungsclustern im Vordergrund der Analyse.

Wichtig ist dabei zu ergründen, welche Techniken sich zur Optimierung von verteilt arbeitendem Code einsetzen lassen, um etwa die Kommunikation zu minimieren. autoBLAS soll jedoch nicht nur verteilt arbeitenden Code generieren, sondern seine aktuellen Vorteile wie die leichte Bedienbarkeit beibehalten. Um diese Ziele zu erreichen, ist zunächst eine umfangreiche Studie der bisherigen Lösungen notwendig, um einerseits State-of-the-Art-Techniken zum Verteilen von Berechnungen und andererseits konkrete Frameworks kennenzulernen, die für eine Implementierung in autoBLAS in Frage kommen.

Generell bietet das Hochleistungsrechnen neben den genannten Herausforderungen bezüglich der Performance noch weitere Aspekte, die es beim Betrieb zu beachten gilt. Dazu gehören etwa der zwischenzeitliche Ausfall von Knoten oder Unterbrechungen des Netzwerks, wodurch es in der Praxis zum Verlust von (Teil-)Ergebnissen kommen kann. Probleme wie diese werden in dieser Arbeit explizit nicht betrachtet. Stattdessen wird angenommen, dass die Implementierung der zugrundeliegenden Schichten eine ausfallsichere Abstraktion bietet beziehungsweise die Berechnung im schlimmsten Fall neu gestartet wird.

1.2 Motivation

autoBLAS bietet seinen Nutzern insbesondere Vorteile in den folgenden Bereichen.

Einfachheit autoBLAS ist einfach zu nutzen und kann daher auch von Personen genutzt werden, die keine oder nur wenige Programmierkenntnisse besitzen. Technische Details werden größtenteils verborgen und eine Fehlbenutzung damit erschwert.

Korrektheit Der generierte Code ist für dieselbe Eingabe immer gleich und ist somit nicht anfällig für Fehler, die aufgrund von Unwissenheit, Unaufmerksamkeit oder Müdigkeit des Programmierers entstehen. Durch die abstrakte Eingabesprache ist die Verifikation der Programmkorrektheit durch Peerreviews vereinfacht.

Entwicklungsgeschwindigkeit Die mathematische Abstraktion, die die Eingabesprache bietet, ist zudem schnell geschrieben und angepasst.

Flexibilität Dadurch, dass autoBLAS-Dateien keine konkreten Implementierung mit einer bestimmten Bibliothek enthalten, sondern davon abstrahiert sind, ist das Auswechseln des verwendeten Backends mit einem einfachen Ändern der Kommandozeilenparameter möglich.

Damit können insgesamt Kosten im gesamten Entwicklungszyklus von Programmen für das Hochleistungsrechnen eingespart werden. In der Planungsphase kann der Fokus auf das zu lösende Problem gelegt werden, während Details der Implementierungssprache nebensächlich sind, was Fehlplanungen vorbeugt. Das Implementieren der Programme kann besser unter den Mitarbeitern aufgeteilt werden, da fachliche Kollegen die autoBLAS-Blöcke mit minimaler Kommunikation mit den Programmierern selbst schreiben können. Dadurch, dass Fachkollegen die autoBLAS-Blöcke selbst verifizieren können, entstehen weniger Fehler durch Fehlkommunikation.

1.3 Aufbau der Arbeit

Das folgende Kapitel bietet einen Überblick über die in dieser Arbeit verwendeteten Konzepte und Begriffe aus dem Bereich des Hochleistungsrechnens. Weiterhin findet sich dort eine tiefergehende Einführung in autoBLAS mit Details zu den unterstützten Operationen, der konkreten Benutzung und aktuellen Defiziten. Die weiteren Teile der Arbeit sind als ein mehrstufiger Filter für existierende Lösungen im Bereich der Verteilung von Berechnungen konzipiert. Diese sind so zahlreich, dass es im Rahmen dieser Arbeit zu aufwändig gewesen wäre, jede von ihnen im Detail zu betrachten. In Kapitel 3 wird zunächst die Vorauswahl vorgestellt, die für diese Arbeit in Frage kommt. Deren Eigenschaften und zugrundeliegende Konzepte werden grob umrissen und mit den Zielen von autoBLAS abgeglichen. Verbleibende Frameworks werden einer detaillierteren theoretischen Analyse unterzogen. Wie diese Kandidaten konkret auf Quellcodeebene zu nutzen sind, wird in Kapitel 4 vorgestellt. In dieser Stufe werden wiederum Frameworks aussortiert, die sich aus praktischen Gesichtspunkten nicht für den Einsatz in autoBLAS eignen. Es bleibt, die Kandidatenframeworks hinsichtlich ihrer Performance zu bewerten. Dies geschieht in Kapitel 5. Schließlich wird vorgestellt, wie die Frameworks, die durch diesen Filter als geeignet befunden wurden, in autoBLAS integriert werden können. Details dazu findet der Leser in Kapitel 6.

2 Grundlagen

Dieses Kapitel gibt eine kurze Einführung in das Hochleistungsrechnen und das Umfeld, in dem autoBLAS angesiedelt ist. Dies dient dem besseren Verständnis der nachfolgenden Arbeit. Weiterhin können die verwendeten Argumente dadurch leichter in einen Gesamtzusammenhang eingeordnet werden.

2.1 Verteilte Systeme

Im Standardwerk *Distributed Systems* von van Steen et al. wird ein verteiltes System definiert als „a collection of autonomous computing elements that appears to its users as a single coherent system“ [86, S. 2]. Ein verteiltes System ist also eine Abstraktion über Rechenressourcen. Seine *computing elements* werden als Knoten bezeichnet. Die genannte Definition ist relativ weit gefasst; entsprechend breit gefächert sind die tatsächlichen Architekturen, Eigenschaften und Einsatzgebiete verteilter Systeme. Dennoch gibt es einige gemeinsame Ziele. Eine typische Motivation für die Einrichtung und Nutzung verteilter Systeme ist die Steigerung der Rechenkapazitäten, etwa, um größere Datenmengen bewältigen zu können. Dabei ist das Verteilen („scaling out“) eine Alternative zur Nutzung immer stärkerer Einzelmaschinen („scaling up“), deren Kosten häufig superlinear mit dem Leistungsvermögen steigen [48, S. 146]. Wichtig ist dabei, dass die Rechenressourcen transparent zwischen den Knoten geteilt werden können, sodass das System insgesamt als Einheit funktioniert. Ein Nutzer sollte nicht wissen müssen, wie viel Speicher auf welchem Knoten zur Verfügung steht und Objekte entsprechend auf bestimmten Knoten anlegen müssen. Im Umkehrschluss bedeutet dies jedoch nicht, dass Nutzer dies nicht wissen *dürfen*; viele Situationen lassen sich mithilfe tieferer Einblicke in ein verteiltes System besser handhaben. Ein Beispiel dafür ist, dass eine gewisse Zeit vergehen kann, bis ein auf einem Knoten erzeugtes Ergebnis auf anderen Knoten sichtbar ist. Mit diesem Wissen kann ein Nutzer die Möglichkeit erwägen, dass seine Informationen veraltet sind. Die für diese Arbeit relevante Art von verteilten Systemen ist diejenige für das Hochleistungsrechnen. Hier bestehen die Systeme häufig aus lokal nah beieinander stehenden Maschinen, die über eine Hochgeschwindigkeitsdatenleitung miteinander verbunden sind [86, S. 26f.]. Zumeist sind diese Maschinen identisch, es gibt jedoch auch häufig spezialisierte Knoten, etwa mit größerem Speicher oder anderer Hardware, etwa GPUs anstelle von CPUs beziehungsweise zusätzlich zu diesen (Abbildung 2.1). Das macht sowohl die für diese Systeme geschriebenen Programme als auch die Zuteilungsalgorithmen zur Laufzeit komplexer, da diese mit Hardwareheterogenität umgehen können müssen. Für das effiziente Speichern und Verteilen von Daten stehen Netzwerkspeicher zur Verfügung, auf die üblicherweise alle Knoten gleichermaßen zugreifen können.

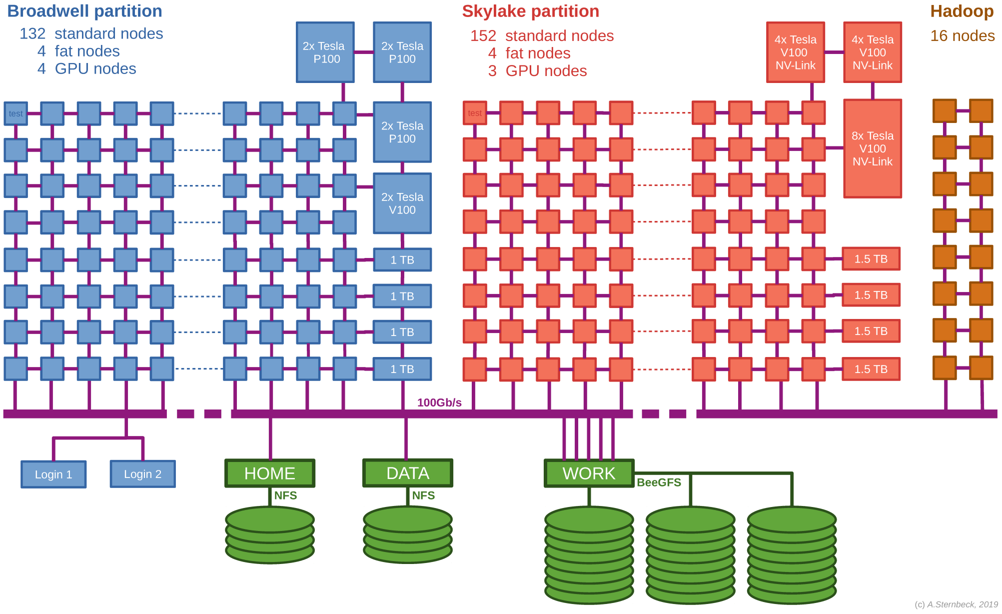

Abbildung 2.1: Beispiel für eine Clusterarchitektur
Quelle: `https://wiki.uni-jena.de/download/attachments/22453005/AraOverview.png`

2.2 Begriffsklärungen

In diesem Abschnitt werden die wichtigsten Konzepte und Begriffe aus dem Bereich des Hochleistungsrechnens erläutert.

SPMD

Single Program, Multiple Data ist ein Programmiermodell, bei dem ein einzelnes Programm gleichzeitig auf mehreren Knoten gestartet wird. Die Arbeitsteilung der Prozesse untereinander geschieht einerseits dadurch, dass jeder von ihnen mit unterschiedlichen Daten arbeitet, und andererseits durch Kommunikationsaufrufe. Besonders häufig ist diese Form des Programmiermodells in High-Performance-Programmen in C/C++ und Fortran durch die beliebte Bibliothek MPI [78] zu finden [19]. Vom Programmierer erfordert dieses Modell eine explizite Aufteilung der Daten sowie der Algorithmen, was allerdings auch eine Chance ist, die einige Vorteile mit sich bringt. Zunächst einmal erhält er damit die volle Kontrolle über die Vorgänge im System, was die Portabilität der Performance verbessert [72]. Weiterhin sind die Compiler und Laufzeitumgebungen im SPMD-Modell tendenziell einfacher aufgebaut, da der Programmierer ihnen einen Teil der schwierigen Arbeit abnimmt [19]. Zuletzt vermeidet diese Methode den Flaschenhals, der dadurch entsteht, nur einen Prozess auf einem Knoten zu starten, der die Arbeit und Daten anschließend auf alle weiteren Knoten verteilen muss [6, 76]. Schließlich hat SPMD aber auch deutliche Nachteile für die Programmierer: Beispielsweise ist es wichtig ausreichende Vorkehrungen zu treffen, damit Probleme wie Deadlocks oder das mehrfache beziehungsweise ausgelassene Berechnen von Teilergebnissen nicht auftreten. Die Klarheit von entsprechend diesem Modell geschriebenem Code ist in der Folge oft durch Details wie Kommunikation oder die Überprüfung von Grenzen beeinträchtigt. Dies schlägt sich auch negativ in der Entwicklungszeit nieder [76, S. 2]. Experten wie Chamberlain et al. sehen hierin „the primary inhibitor of of parallel programmability today (Anm.: 2007)“ begründet [19, S. 294].

Task

Ein Task ist eine Ausführungseinheit für Code und damit mit einer Funktion vergleichbar, jedoch mit einem stärkeren Bezug auf parallel arbeitende Systeme. Entsprechend sind Tasks dafür konzipiert, nebenläufig zueinander ausgeführt zu werden, wobei Abhängigkeiten zwischen Tasks existieren können, die eine komplett freie Ausführung von Tasks einschränken. Tasks werden daher häufig in Abhängigkeitsgraphen organisiert, worauf im späteren Teil der Arbeit bei der Diskussion der parallelen Frameworks genauer eingegangen wird.

NUMA

Non Uniform Memory Access ist ein Architekturmodell, bei dem Knoten sowohl auf eigenen Speicher zugreifen können, aber auch auf den anderer Knoten. Innerhalb von Programmen lassen sich Abstraktionen einsetzen, die die Speicherzugriffe unabhängig vom Ziel im Quellcode gleich aussehen lassen. Die Laufzeitcharakteristiken, insbesondere die Speicherzugriffszeiten, unterscheiden sich jedoch deutlich zwischen lokalen und entfernten Zugriffen [29, S. 2]. Die später vorgestellten Speicherzugriffsmodelle bieten verschiedene Ansätze für Software, um mit diesen Möglichkeiten der Hardware umzugehen.

BLAS

Eine tiefergehende Diskussion der Merkmale von autoBLAS ist nicht möglich, ohne zuvor die BLAS kurz vorzustellen. Die BLAS-Schnittstelle (*Basic Linear Algebra Subprograms*) ist eine der wichtigsten im High-Performance-Computing, die auf den Vorschlag von Hanson et al. [40] zurückgeht und initial 1979 implementiert wurde [54]. Sie wurde seitdem mehrmals erweitert, etwa 1988 um die sogenannten Level-2-Funktionen [31] und 1990 um die Level-3-Funktionen [30]. Darüber hinaus existieren zahlreiche Reimplementierungen wie GotoBLAS [36, 37], OpenBLAS [90] und cuBLAS [26] sowie darauf basierende Erweiterungen wie LAPACK [2], ATLAS [88], Intel MKL [44] und Eigen [39]. Weiterhin erwähnenswert sind die auf verteiltes Rechnen ausgelegten Erweiterungen PBLAS [24] und ScaLAPACK [23]. Wie der Name bereits andeutet, definieren die BLAS Methoden für zentrale und grundlegende Aufgaben der linearen Algebra, die von großer Bedeutung für viele Anwendungen im High-Performance-Computing sind. Mithilfe dieser lässt sich eine ganze Bandbreite an Problemen formulieren, die relevant in Wirtschaft, Wissenschaft und Technik sind. Zu ihnen zählen vor allem das maschinelle Lernen und Datamining – darunter insbesondere das Deep Learning – sowie vielfältige Vorhersageaufgaben in der Wirtschaft und in der Meteorologie. Weiterhin basieren bekannte Algorithmen wie PageRank zum Sortieren von Suchergebnissen im Internet auf Problemen der linearen Algebra, in diesem Fall einem Eigenwertproblem [65]. Die zuvor genannten BLAS-Level beziehen sich auf die Komplexität der unterstützten Algorithmen [30, S. 2]. Der originale BLAS-Vorschlag, der später zum Level 1 erklärt wurde, umfasst Vektor-Vektor-Operationen wie Skalarprodukte, Additionen, Normen, Kopien und Swaps [40]. Das zweite BLAS-Level legt den Fokus auf Matrix-Vektor-Operationen und unterstützt beispielsweise Matrix-Vektor-Produkte und die Lösung linearer Gleichungssysteme mit Dreiecksmatrizen [31]. Im Level 3 sind verschiedene Matrix-Matrix-Operationen (etwa Addition und Multiplikation) sowie die Lösung von linearen Gleichungssystemen mit mehreren rechten Seiten enthalten [30].

Die BLAS sind zweifelsohne zentral für zeitgemäßes wissenschaftliches Rechnen [90]. Dennoch ist der Zugang zu deren vollem Potenzial tendenziell Programmen vorbehalten, die in den Hochleistungssprachen C, C++ und Fortran geschrieben sind, was sich auch in der vorliegenden Arbeit bestätigen wird. Diese geben dem Programmierer ein hohes Maß an Kontrolle über die Hardware, was eine fast maximale Performance ermöglicht, allerdings auch Potenzial für Fehler birgt. Zudem hat nicht jeder, der Bedarf an der schnellen Berechnung von Ausdrücken der linearen Algebra hat, ausreichende Kenntnisse dieser Sprachen, um BLAS effektiv nutzen zu können. Eine mögliche Lösung dafür ist die Nutzung von Sprachen, die zwar höher in der Abstraktionshierarchie liegen, aber dennoch den effizienten BLAS-Code nutzen können, etwa Python mit NumPy [41] oder Julia [11]. Ein weiteres Problem ergibt sich aus der Vielzahl der BLAS-Implementierungen und einem potenziellen Lock-In. Ein Programm, das beispielsweise mit Eigen geschrieben wurde, kann nicht ohne größeren Aufwand auf cuBLAS umgestellt werden, sollten es die Umstände erfordern. Dies erschwert unter anderem das Testen, welche Implementierung sich in einem konkreten Szenario am meisten eignet.

2.3 autoBLAS

autoBLAS entstand im Umfeld des GENO-Projekts[1], zu dem außerdem Matrix Calculus[2] gehört. Zusammen bieten sie Nutzern insbesondere die Möglichkeit, Lösungsprogramme (sogenannte Solver) für Optimierungsprobleme mit Bezug zur linearen Algebra auf eine einfache und dennoch effiziente Art zu implementieren. Sie alle teilen sich eine ähnliche Eingabesprache, die nah an der mathematischen Formulierung liegt und sich damit an beispielsweise MATLAB oder Octave orientiert.

Matrix Calculus ist auf die Ableitung von Tensorausdrücken spezialisiert. Intern wurde dafür zunächst der Ricci-Kalkül verwendet [52] und später die Einstein'sche Summenkonvention unterstützt [51]. Nach Eingabe eines abzuleitenden Ausdrucks kann der Nutzer ein Pythonprogramm herunterladen, das die Ableitung umsetzt. Der Nutzer kann Matrix Calculus zusätzliche Metadaten zur Verfügung stellen, um das Ergebnis zu verbessern. So lassen sich beispielsweise Ausdrücke vereinfachen und entsprechend das generierte Programm effizienter implementieren, wenn für eine gegebene Matrix bekannt ist, dass sie symmetrisch ist.

---

[1] `http://geno-project.org`

[2] `http://matrixcalculus.org`

Listing 2.1: Lineares Optimierungsproblem beschrieben in GENO

```
parameters
  Matrix A
  Vector b, c
variables
  Vector x
min
  c'*x
st
  A*x <= b
  x >= 0
```

GENO ist ein Framework für generische Optimierung, mit dessen Hilfe sich Solver für ein breites Spektrum an Optimierungsproblemen generieren lassen [53]. Ein Beispiel für ein klassisches lineares Optimierungsproblem ist in Listing 2.1 abgebildet. Ausgehend davon kann ein Pythonprogramm generiert werden, das bestehende Pythonbibliotheken zum Lösen verwendet oder auf einen spezialisierten GENO-Solver zurückgreift. GENO ermöglicht es damit, hocheffiziente Solver für Probleme zu generieren, die nicht Teil einer Standardbibliothek sind. Es bietet dem Nutzer damit ähnliche Vorteile wie autoBLAS im Bereich der Entwicklungsgeschwindigkeit, Korrektheit und Einfachheit.

Übersicht autoBLAS autoBLAS ist ein Transpiler, das heißt ein Programm, das Quellcode in anderen Quellcode transformiert. Als Teil des GENO-Projekts hatte es von Beginn an die Aufgabe, die Berechnung von Ausdrücken linearer Algebra effizient zu implementieren. Dafür unterstützt autoBLAS verschiedene Technologien und Programmiersprachen im BLAS-Umfeld, mit denen Programme generiert werden können. Sofern es nicht anders angegeben wird, ist mit autoBLAS die Technologie beziehungsweise das Konzept gemeint, nicht die gleichnamige Referenzimplementierung. Letztere ist aufgrund zahlreicher Bugs nicht für den Praxiseinsatz geeignet, sondern lediglich für Forschungszwecke.

Mit autoBLAS können Ausdrücke der linearen Algebra in einer einfachen Syntax beschrieben werden, die durch MATLAB einem breiten Personenkreis auch außerhalb der High-Performance-Community bekannt ist (Listing 2.2). Diese Ausdrücke können in eine Quellcodedatei eingebettet werden, indem sie mit `#autoblas`-Blöcken umgeben werden. Innerhalb dieser Bereiche dürfen Variablen deklariert und Zuweisungen beschrieben werden. Aus diesen Ausdrücken generiert autoBLAS anschließend effizienten Code. Der Nutzer hat die Wahl zwischen verschiedenen sogenannten Backends, auf die später noch eingegangen wird. Im Wesentlichen sind sie eine Möglichkeit, schnell und einfach die konkrete Implementierung der BLAS auszuwechseln. Für Anwender ergeben sich damit mehrere Vorteile im Bereich der Entwicklungsgeschwindigkeit, Korrektheit und Flexibilität. Mit der Nutzung von autoBLAS vergeht weniger Zeit von der Idee bis zur produktionsreifen Umsetzung, da der Code kürzer ist und es nicht notwendig ist, sich mit Details auseinanderzusetzen, wie etwa dem Speichermanagement. Durch die kürzere und direktere Form ist der Code einfacher zu lesen und zu verstehen, was eine leichtere Überprüfung der Korrektheit erlaubt.

Listing 2.2: Beispielinput für autoBLAS

```
#autoblas {
  Vector y;
  Vector w;
  Vector g f=c data=gPtr size=N stride=1;
  y = w .* g;
}
```

Listing 2.3: Output von autoBLAS mit dem MKL-Backend

```
vdMul(y.size(), w.data(), gPtr, y.data());
```

Listing 2.4: Output von autoBLAS mit dem Eigen-Backend

```
{
Eigen::VectorXd::ConstMapType g(gPtr, N);
y = (w.array() * g.array()).matrix();
}
```

Insbesondere wird auch den Fachkollegen, die keine Programmierkenntnisse besitzen, eine Beteiligung an Peer-Review-Prozessen ermöglicht. Die Generierung von Quellcode hat den Vorteil, dass keine Fehler durch menschliche Unachtsamkeit in den Details des Programms verursacht werden können. autoBLAS folgt damit der bewährten Unix-Philosophie [67, S. 45].

Zwei Beispielausgaben von autoBLAS für verschiedene Backends innerhalb von C++ sind in den Listings 2.3 (MKL-Backend) und 2.4 (Eigen-Backend) abgebildet. Eine Auswahl des Backends erfolgt über Kommandozeilenargumente, wobei zum aktuellen Zeitpunkt Eigen, MKL, NumPy und SQL mit inbegriffen sind. Dies verdeutlicht die Flexibilität von autoBLAS, die ein Umschalten zwischen den konkreten Backends ohne Aufwand ermöglicht. Auch ein Umsteigen auf neue Technologien ist mit diesem Modell problemlos möglich, indem ein neues Backend zu autoBLAS hinzugefügt wird. Ein Anpassen aller alten Anwendungen ist nicht notwendig, wenn jene autoBLAS-Anweisungen verwenden. Den gleichen Vorteil bieten auch abstrahierende Bibliotheken, indem sie intern eine neue Implementierung verwenden. Der Vorzug, den autoBLAS gegenüber diesem Ansatz hat, ist die Möglichkeit zum Finetuning. Auch wenn der hauptsächliche Anwendungsfall der Einsatz als Präprozessor vor einem Compiler oder Interpreter ist, besteht die Möglichkeit, die Ausgabe von autoBLAS weiterzuverwenden und manuell zu bearbeiten.

Trotz der Stärken hat das autoBLAS-Konzept auch einige Nachteile. Der offensichtlichste von ihnen ist die Einschränkung auf einzelne, voneinander unabhängige Blöcke. Für autoBLAS ist es weder möglich, Bezüge zwischen den Blöcken herzustellen[3] noch den Kontext der Blöcke miteinzubeziehen. Dadurch fehlen autoBLAS globale Informationen, die es zur Optimierung der Ausgabe verwenden könnte. Zu diesen Informationen gehört zum Beispiel, ob sich ein Block innerhalb einer Schleife befindet. Auf dieses Problem wird innerhalb dieser Arbeit noch mehrmals Bezug genommen. Ein weiteres, prinzipielles Problem ist die Nichtentscheidbarkeit für sta-

---

[3] Für Blöcke innerhalb einer Datei wäre dies möglich, jedoch nicht darüber hinaus.

Tabelle 2.1: Übersicht der von autoBLAS unterstützten Operationen mit jeweiligem Anwendungsbereich (Skalare, Vektoren oder Matrizen).

Symbol(e)	Operation	Anwendungsbereich
`+`, `-`	Addition, Subtraktion	S, V, M
`*`	Multiplikation	paarweise S, V, M
`^`, `/`	Potenzierung, Division	S
`.*`, `./`	elemw. Multiplikation, Division	S, V, M
`'`	Transposition	S, V, M
`log(..)`	elemw. natürlicher Logarithmus	S, V, M
`exp(..)`	elemw. Exponentialfunktion	S, V, M
`Vector(Val, Size)`	Erzeugen eines Vektors	—
`rows(..)`, `cols(..)`	Abfrage von Dimensionen	S, V, M in `Vector`
`size(..)`	Abfrage der Länge	V in `Vector`

tische Programmanalysierer. Dies bedeutet, dass „it is impossible to build an analysis that would correctly decide a property for any analyzed program“ [61, S. 4]. Dies schließt das bekannte Halteproblem mit ein, ist aber nicht darauf beschränkt. Unter anderem ist es auch unmöglich, alle Codeteile für alle Programme zu identifizieren, die parallelisierbar sind [55, S. IV].

Technische Details von autoBLAS Die Deklaration von Variablen beginnt mit einem Typen (`Matrix`, `Vector` (Spaltenvektor) oder `Scalar`), der von einem Namen aus alphanumerischen Zeichen und Unterstrichen gefolgt wird. Anschließend folgt eine beliebig lange Liste an Metadaten, etwa die Dimensionalität oder der Stride.

Manche Backends erfordern technische Metadaten, etwa einen Pointer zum Start des Vektors oder der Matrix, wie in den MKL- und Eigen-Beispielen zu sehen ist. Mit autoBLAS’ Frontend-Parameter kann angegeben werden, dass die benannten Variablen bereits ein bestimmtes Format haben. Ist es mit dem Backend kompatibel, müssen die Deklarationen nicht zwingend in den Code übernommen werden. Weiterhin lässt sich in autoBLAS für jede einzelne Variable mittels einer Spezifikation des `f`-Metadatums angeben, dass sie noch nicht den richtigen Datentyp hat. Beispielsweise wurde in den Listings 2.3 und 2.4 das Eigen-Frontend gewählt. Das MKL-Backend unterstützt dieses direkt, indem es versteht, wie es auf die rohen Daten der Eigen-Datentypen zugreifen kann. autoBLAS nimmt für die Variablen `y` und `w` an, dass sie bereits in Eigen definierte Vektoren sind, definiert `g` aber aufgrund der Angabe des C-Frontendes mittels `f=c` als einen passenden Eigen-Datentypen. Damit bei dieser Deklaration einer Variable keine Compilerfehler wegen mehrfach deklarierter Variablen auftreten, führt autoBLAS einen neuen Block und damit Namensraum ein. autoBLAS trifft dazu die Annahme, dass die eingesetzten Datentypen keinen Destruktor haben, der den Speicherbereich freigibt, auf den der Eingabepointer zeigt. Anderenfalls kann es zu Use-After-Free-Fehlern kommen, die im besten Fall das Programm zum Absturz bringen, in schlimmeren Fällen aber falsche Ergebnisse oder Sicherheitslücken erzeugen können.

In Bezug auf Statements unterstützt die autoBLAS-Referenzimplementierung derzeit ausschließlich Zuweisungen von Ausdrücken zu Variablen. In Ausdrücken sind dabei die Operationen erlaubt, die in Tabelle 2.1 dargestellt sind. Die meisten davon sind in jeder Bibliothek für lineare Algebra vorhanden und müssen daher nicht weiter erläutert werden. Der Operator `Vector(Val, Size)` erzeugt einen Vektor der Länge `Size`, wobei jedes Element den Wert `Val` erhält. Für `Size` können keine konkreten Werte eingetragen werden, sondern lediglich relative Werte mittels der Operatoren `rows(..)`, `cols(..)` und `size(..)`. Diese können wiederum nicht überall im autoBLAS-Block verwendet werden, sondern ausschließlich innerhalb des `Vector(..)`-Aufrufes. Zudem können sie lediglich Variablennamen als Argumente annehmen; Ausdrücke wie `rows(b')` sind nicht erlaubt. Wichtige Operationen wie `sum`, `norm1` und `norm2` sind momentan ebenfalls nicht in autoBLAS implementiert. Bei Bedarf lassen sich die Identitäten $sum(x) = \mathbf{1}^T x$ sowie $norm2(x) = \exp(\log(\mathbf{1}^T(x. * x))/2)$ verwenden, wobei $\mathbf{1}$ in diesen Formeln für `Vector(1, size(x))` steht.

Ein weiteres Detail, das bisher unerwähnt geblieben ist, ist die `#autoblas-include`-Anweisung. Diese führt dazu, dass autoBLAS alle Importanweisungen generiert, die für die Nutzung des jeweiligen Backends notwendig sind. Dies kann dazu führen, dass manche davon doppelt im Programm erscheinen, wenn sie bereits manuell vom Programmierer spezifiziert worden sind. In der Regel stellt dies jedoch kein Problem dar, da mehrfache Importanweisungen üblicherweise ignoriert werden, etwa durch Include-Guards.

Wie bereits erwähnt, hat die Referenzimplementierung einige Bugs, die zum Beispiel zu unerwarteten Programmabstürzen führen. Für diese Arbeit von größerer Bedeutung ist jedoch der Bug, dass in einem Block deklarierte Variablen für autoBLAS im ganzen Input gültig sind. Dies führt dazu, dass die gleichen Namen nicht in mehreren Blöcken verwendet werden können, da sonst ein Fehler wegen Mehrfachdeklarationen auftritt. Ein praktisches Problem stellt dies dar, wenn verschiedene Front- und Backends verwendet werden, die eine Konvertierung erfordern, etwa das C-Frontend und das Eigen-Backend. In diesem Fall wird die Deklaration der Eigen-Variablen nur im ersten Block generiert; in allen anderen wird der nachfolgende Compiler oder Interpreter deshalb die in den Formeln verwendeten Variablen nicht finden.

3 Analyse bestehender Lösungen

Das Verteilen von Berechnungen jeglicher Art ist in Softwaresystemen das Mittel der Wahl, um die Arbeitslast in den verschiedensten Anwendungsgebieten bewältigen zu können. Dabei handelt es sich keinesfalls um ein neues Problem. Bereits seit mehreren Jahrzehnten forschen Industrie und Wissenschaft in diesem Bereich und haben in diesem Zeitraum eine schwer zu überschauende Menge an unterschiedlichen Lösungen hervorgebracht. Unter ihnen finden sich diverse Ansätze, Ideen, Ziele und Implementierungen und damit auch Eigenschaften. Die meisten dieser Ansätze haben keine weite Annahme außerhalb von Modellanwendungen gefunden, einige weitere wurden eine Zeit lang eingesetzt, aber dann von neuen Systemen abgelöst. Nur wenige Lösungen haben sich als verlässlich, effizient und benutzerfreundlich genug herausgestellt, um eine lang andauernde und weite Nutzung zu erfahren, wie es beispielsweise dem *Message Passing Interface* (MPI) gelungen ist.

In diesem Kapitel werden einige wichtige und einflussreiche Beiträge untersucht und deren Merkmale vorgestellt, darunter sowohl ältere, initiale Arbeiten als auch moderne Ansätze. Es werden sowohl Arbeiten speziell aus dem Umfeld der linearen Algebra betrachtet als auch aus dem verteilten Rechnen im Allgemeinen. Die Untersuchung erfolgt auf einer allgemeinen Ebene und berücksichtigt etwa die konzeptuelle Ausrichtung und daraus resultierende Eigenschaften, jedoch noch nicht eine mögliche Implementierung. Damit werden zweierlei Ziele verfolgt. Einerseits wird damit effizient eine Vorauswahl von Kandidatenframeworks getroffen, die sich als Backends für das verteilende autoBLAS eignen, ohne Rücksicht auf Implementierungsdetails nehmen zu müssen. Jedes Framework auf dieser Analyseebene zu untersuchen wäre zu aufwändig und nicht zielführend, wenn es bereits zu große konzeptuelle Unterschiede gibt. Weiterhin werden die den Frameworks zugrundeliegenden Algorithmen und Ansätze zum Verteilen von Berechnungen hervorgehoben, um daraus Schlussfolgerungen für die autoBLAS-Implementierung zu ziehen. Wichtige Informationen betreffen etwa bewährte Programmiermodelle, aber auch häufige Fallstricke. Um von diesen Erkenntnissen profitieren zu können, ist es wichtig, den Blick nicht auf Arbeiten im Bereich der linearen Algebra einzuschränken.

Zur Erleichterung eines strukturierten Vergleichs braucht es eine Auswahl an Kategorien, entsprechend derer die bisher entwickelten Frameworks geordnet und gruppiert werden können. In dieser Arbeit wird dafür die Taxonomie von Thoman et al. verwendet. Im Folgenden wird diese zunächst kurz erläutert. Anschließend erfolgt die Vorstellung und Untersuchung der Frameworks in zwei Stufen. Zunächst wird der Großteil überblicksartig aufgezeigt, bevor anschließend einige besonders wichtige Arbeiten detaillierter präsentiert werden.

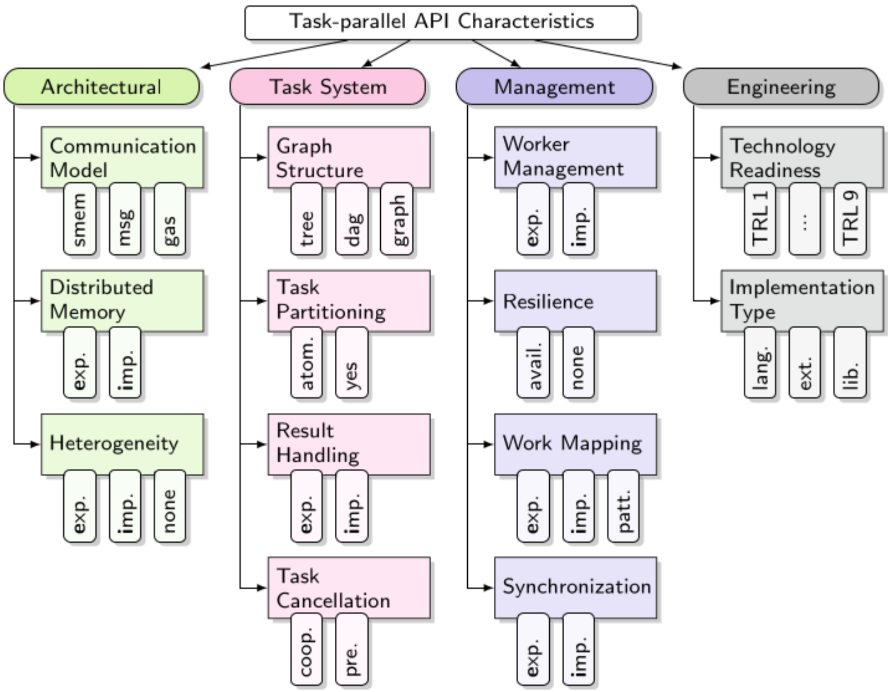

Abbildung 3.1: Taxonomie der Programmierschnittstelle nach Thoman et al.
Quelle: [81, S. 1425]

3.1 Taxonomie paralleler Programmierschnittstellen

Die in diesem Kapitel beschriebene Taxonomie von Thoman et al. wurde gewählt, da sie sowohl umfassend als auch prägnant ist [81]. Weiterhin ist sie erst vor relativ kurzer Zeit veröffentlicht worden und damit in der Lage, auch moderne Entwicklungen abzubilden. Insgesamt wird durch sie eine einfache Vergleichbarkeit der Programmierschnittstellen und der Laufzeitumgebung zwischen verschiedenen Ansätzen im High-Performance-Computing ermöglicht. Die relevanten Eigenschaften der Programmierschnittstelle sind zunächst in vier große Kategorien gegliedert: Architektur, Tasksystem, Management und technische Aspekte (Abbildung 3.1). Diese bilden die Ecksteine dafür, welche Aufgaben dem Programmierer abgenommen werden und welche Eingriffs- und Optimierungsmöglichkeiten ihm bleiben. Im Folgenden werden die wichtigsten Merkmale kurz vorgestellt, um im weiteren Teil der Arbeit deren Implikationen bewerten zu können.

Das Kommunikationsmodell kann die Ausprägungen *Shared Memory*, *Message Passing* oder *Global Address Space* (GAS) annehmen. In Shared-Memory-Systemen kann jeder Knoten auf jeden Speicherbereich im Adressraum des Systems zugreifen, als wäre er lokal vorhanden. Dieses Modell stellt die Implementierung einer Problemlösung in den Mittelpunkt und vermeidet explizite Kommunikationsanweisungen [1]. Nichtsdestotrotz wird dieses Modell unter anderem aufgrund von Skalierungsproblemen für neuere Systeme kaum mehr in Betracht gezogen [19, S. 301]. Bei nachrichtenbasierter Kommunikation sind die Speicherbereiche der einzelnen Knoten komplett voneinander getrennt. Ein Austausch von Daten ist nur über den expliziten Austausch von Nachrichten möglich [4]. Dies bedeutet zwar mehr Aufwand für den

Programmierer, erlaubt andererseits aber auch eine bessere Optimierung durch den Compiler, da die verarbeiteten Daten lokal vorliegen [28]. GAS, auch PGAS (*Partitioned Global Address Space*) genannt, ist ein Kommunikationsmodell, bei dem jeder Knoten seinen eigenen Speicherbereich besitzt, auf den er direkt zugreifen kann, aber zusätzlich auch mittels bestimmer Funktionsaufrufe Daten von anderen Knoten lesen und schreiben kann. Sein Vorteil gegenüber nachrichtenbasierter Kommunikation liegt im reduzierten expliziten Synchronisationsaufwand, allerdings auf Kosten zusätzlicher Fehlerquellen [16].

Eine implizite Verteilung der Daten bietet dem Programmierer mehr Komfort und eine tendenziell besser überprüfbare Korrektheit des Programms. Dazu zählt beispielsweise, dass Daten, auf die von einem Knoten aus zugegriffen wird, vor dem Zugriff gegebenenfalls kopiert werden, sodass sie für den Knoten in jedem Fall verfügbar sind. Dies nimmt möglicherweise die Kontrolle über die Ausführung, die notwendig ist, um die Performance zu maximieren.

Die Heterogenität bezieht sich auf die Unterstützung verschiedener Hardwarekomponenten wie GPUs oder FPGAs [87] innerhalb des Clusters. Bei implizit unterstützter Heterogenität kann das System selbstständig auswählen, welche Programmanweisungen auf welcher Hardware ausgeführt werden sollen. Anderenfalls muss der Programmierer diese Verteilung selbstständig vornehmen, wie es etwa beim CUDA-Programmiermodell der Fall ist [25].

Die Graphenstruktur des Tasksystems legt fest, in welcher Form Tasks voneinander abhängen dürfen. Relevant sind insbesondere Baumstrukturen und kreisfreie, gerichtete Graphen (DAG), da sie das Management der Tasks gegenüber allgemeinen Graphen deutlich vereinfachen, etwa bei der Entscheidung, wann die Ressourcen der Tasks freigegeben werden können oder welche Tasks als nächstes ausgeführt werden, weil in diesem Fall nur die aktuelle „Front“ betrachtet werden muss [15].

Das Result-Handling ist implizit, wenn der Nutzer direkt mit den problembezogenen Datentypen arbeitet, und explizit, wenn die Daten in berechnungsbezogene Datentypen wie Futures gepackt sind. Die erste Variante ist einfacher in der Nutzung, ignoriert jedoch die grundlegenden Unterschiede zwischen normalen Funktionsaufrufen und der Kommunikation über das Netzwerk [48, S. 134]. Futures ermöglichen eine Abfrage des Berechnungszustandes und können besser mit Situationen umgehen, in denen die Berechnung aus unerwarteten Gründen fehlgeschlagen ist, etwa weil die Netzwerkverbindung unterbrochen wurde.

Das Workermanagement umfasst die Erzeugung, Konfiguration und Beendung von Workern, das heißt Threads oder Prozesse. Dieser Aspekt ist bei den meisten Systemen implizit implementiert [81].

Der Aspekt des Work-Mappings beschreibt den Schedulingprozess, das heißt, welcher Prozessor zu welchem Zeitpunkt welche Arbeit zugewiesen bekommt. Eine effiziente Methode hierfür ist sogenanntes Work-Stealing, bei dem jeder Prozessor eine Warteschlange mit den nächsten anstehenden Aufgaben verwaltet. Ist die Warteschlange eines Prozessors leer, übernimmt er Aufgaben von einem anderen Prozessor [13]. Die Alternative dazu ist Work-Sharing, wobei ein eigener Schedulerprozess konstant versucht, die Tasks so umzulagern, dass jeder Prozessor zu jedem Zeitpunkt ausreichend Arbeit hat.

Besonders wichtig bei jeder verteilten Berechnung ist die Synchronisation. Für eine explizite Synchronisation existieren zahlreiche Standardtechnologien und -algorithmen, etwa Mutexe oder Barrieren. Deren Aufwand kann eingespart werden, sollte sich das Programm implizit synchronisieren lassen, beispielsweise indem jeder Prozessor ein unabhängiges Teilproblem löst.

Der Implementierungstyp lässt sich grob in Bibliotheken, Spracherweiterungen und komplett selbstständige Sprachen unterteilen. Bibliotheken haben den Vorteil, dass sie direkt von der bestehenden Toolchain unterstützt werden und damit einen relativ geringen Mehraufwand (innerhalb dieses Vergleichs) zur eigentlichen Funktionalität der Bibliothek sowohl für den Implementierer der Bibliothek als auch den Nutzer bedeuten. Dem gegenüber steht die Einführung einer komplett eigenen Sprache. Dies hat den Vorteil für den Implementierer, dass diese Sprache speziell auf seine Bedürfnisse zugeschnitten werden kann. Beispielsweise ermöglicht das im vorliegenden Bereich des verteilten Rechnens, relevante Informationen wie Abhängigkeiten zwischen den verschiedenen Aufgaben oder eine Speicherhierarchie zu kodieren. Beides unterstützt die Verteilungsentscheidungen und kann zusätzliche Sicherheit etwa gegen Wettlaufsituationen bieten. Andererseits bedeutet dies einen sehr viel höheren Aufwand, da auch grundlegende Algorithmen für diese Sprache neu geschrieben oder zumindest portiert werden müssen, da die Sprache sonst an Praxistauglichkeit einbüßt. Weiterhin wird ein Compiler oder Interpreter benötigt, um in der neuen Sprache geschriebene Programme überhaupt ausführen zu können. Für eine Benutzbarkeit in der Praxis reicht dies jedoch noch nicht aus. Hinzu kommen nämlich noch weitere relevante Bestandteile moderner Toolchains, etwa Debugger oder Profiler [57]. Projekte wie LLVM [50] beschleunigen diesen Prozess zwar, dennoch ist diese Methode besonders zeitintensiv. Spracherweiterungen bieten einen Mittelweg, der je nach Anforderungen eher in die eine oder andere Richtung ausgelegt werden kann. Beispiele hierfür sind etwa die Einführung neuer Schlüsselwörter oder spezieller Makros.

3.2 Literaturüberblick

Für diese Arbeit wurden insgesamt mehr als 35 Lösungen beziehungsweise Frameworks für verteiltes Rechnen im High-Performance-Kontext untersucht. Die Begriffe „Lösung“ und „Framework“ werden in vielen Teilen dieser Arbeit synonym verwendet, jedoch bezieht sich ersterer Begriff eher auf die theoretische Grundlegung, während letzterer eher die daraus resultierenden Softwareprodukte meint. Dabei ist selbst der Begriff des Softwareprodukts nicht ganz scharf definiert, da zum Beispiel einige Frameworks mehrere Schnittstellen besitzen, durch die auf sie zugegriffen werden kann. Dies geschieht insbesondere, wenn die Frameworks in zunächst einem Bereich sehr erfolgreich sind und sich in der Folge ganze Ökosysteme an Bibliotheken und Sprachen entwickeln, die mit dem Grundframework interagieren. Ein typisches Muster ist beispielsweise, dass ein Framework mit einer Laufzeitumgebung in C++ programmiert wird und dort eine entsprechende Schnittstelle anbietet, zur leichteren Benutzung jedoch mindestens eine Pythonbibliothek nachgereicht wird und in manchen Fällen sogar eine komplett eigene Sprache [6, 60, 82]. Alle untersuchten Lösungen wurden für diese Arbeit basierend auf ihren Merkmalen grob in die Kategorien Allzweckkommunikationsbibliotheken, frühe Ansätze, Pythonbibliotheken und komplexe Softwaresysteme gegliedert, um ähnliche Ansätze gemeinsam betrachten und vergleichen zu können.

3.2.1 Allzweckkommunikationsbibliotheken

Im Bereich der grundlegenden Kommunikation sind wie zuvor erwähnt insbesondere zwei Modelle verbreitet: das nachrichtenbasierte und das PGAS-Modell. MPI ist der Standard für nachrichtenbasierte Kommunikation [19, S. 298]. Es wird aus diesem Grund später in Abschnitt 3.3.1 näher betrachtet. Daneben gibt es den Ansatz PVM (*Parallel Virtual Machines*), der insbesondere in den Neunzigerjahren populär war [77]. PVM bietet eine höhere Resilienz gegenüber Ausfällen, aber generell eine schlechtere Performance als MPI und hat in der Folge mit der Jahrtausendwende deutlich an Einfluss verloren [35]. Daneben gibt es einen größeren Aufschwung im PGAS-Bereich. Im Zentrum von PGAS stehen einseitige Lese- und Schreiboperationen (*Remote Memory Access*; RMA), mit denen explizite Synchronisation zwischen den Kommunikationsteilnehmern eingespart werden kann. Der Vorteil dessen für das Hochgeschwindigkeitsrechnen ist, dass RMA in Clustern häufig direkt von der Hardware unterstützt wird und damit besonders effizient ist [62, S. 233] Insgesamt bietet sich damit ein asynchroner Kommunikationsablauf, bei dem interessierte Knoten benachrichtigt werden, wenn Daten vollständig versendet wurden. Für optimale Performance muss die asynchrone Kommunikation allerdings nicht nur konzeptuell, sondern auch durch die Hardware umgesetzt werden. Beispielsweise ist der Vorteil der Verschränkung des Datenempfangs mit einer Berechnung verringert, wenn die CPU extra Zyklen und womöglich Platz im Cache für den Datenempfang verwenden muss [62]. Wichtige PGAS-Kommunikationsbibliotheken sind GASNet-EX [14], GASPI [38], ARMCI [62] und OpenSHMEM [21]. Eine weiterführende theoretische Untermauerung von PGAS haben Calin et al. beschrieben. Im direkten Vergleich zeigen sich einige Unterschiede zwischen den einzelnen PGAS-Bibliotheken. Entsprechend ergeben sich auch verschiedene Anwendungsgebiete. GASNet-EX und ARMCI bieten hierarchieniedrigere Kommunikationsfunktionen an, die sich eher an Implementierer von Laufzeitsystemen richten. Entsprechend findet GASNet-EX Verwendung in vielen der nachfolgend beschriebenen Systeme für verteiltes Rechnen, etwa Chapel und Legion [14]. Als zusätzliche Einschränkung unterstützt ARMCI ausschließlich homogene Umgebungen [62, S. 233]. OpenSHMEM hingegen ist ein Standard, der viele zuvor existierende SHMEM-Bibliotheken in einer gemeinsamen Schnittstelle vereinigt und sich eher an Anwendungsentwickler richtet. Weitere Unterschiede liegen in den Garantien, die die verschiedenen Bibliotheken bieten. Die hier genannten Bibliotheken geben den Anwendern beispielsweise im Allgemeinen keine Garantie darüber, in welcher Reihenfolge zwei asynchrone Anfragen abgearbeitet werden. Teilweise gibt es jedoch Barrieren zum Erzwingen der Reihenfolgen und leichtere Garantien, etwa, dass zwei Anfragen an denselben Knoten in der richtigen Reihenfolge bearbeitet werden.

Zwischen den Kommunikationsbibliotheken besteht generell eine gewisse Interoperabilität. Da PGAS eine Mischung aus nachrichtenbasierter Kommunikation und geteiltem Speicher darstellt, ist häufig eine Mischung von PGAS-Bibliotheken und MPI möglich, um die Vorteile beider Ansätze zu nutzen [59, 62]. Darüber hinaus übernimmt auch MPI vermehrt Einflüsse aus dem PGAS-Umfeld, etwa asynchrone Kommunikationsaufrufe und einseitige Kommunikation [79].

3.2.2 Frühe Ansätze

Basierend auf den zuvor genannten Kommunikationsbibliotheken haben sich bereits früh ausdrucksstarke Spracherweiterungen und Bibliotheken für verteiltes Rechnen herausgebildet. Aufgrund der Zeit sind diese Bibliotheken häufig in Fortran oder C geschrieben. Obwohl das Hochleistungsrechnen damals noch verhältnismäßig neu war [73], gab es bereits so viele Lösungsansätze für verteiltes Rechnen, dass sie nicht einmal grob in dieser Arbeit besprochen werden können. Stattdessen wird sich hier auf eine Auswahl konzentriert, die auch von aktuellen Arbeiten noch häufiger referenziert wird. Zu ihnen zählen C* [80], Split-C [27], High Performance Fortran (HPF; [69]), Global Arrays [63], NCX [92], Cilk [12], Titanium [91], Co-Array Fortran (CAF; [64]) und Unified Parallel C (UPC; [17]). Hier sind bereits bis heute wichtige Konzepte umgesetzt worden. Dazu zählt beispielsweise die Entscheidung, ob Daten und/oder Programme parallelisiert werden. NCX unterstützt etwa keine parallelen Datentypen. Der Global-Arrays-Ansatz hingegen ist genau darum zentriert. Die automatische Extraktion von potenziell parallel ausführbaren Programmteilen war noch nicht weitgehend erforscht [92]. Aus diesem Grund setzen diese frühen Ansätze häufig auf eine explizite Auszeichnung der parallelisierbaren Codeteile. Allerdings findet sich dieser Ansatz auch etwa 20 Jahre später in den modernen, großen Systemen wieder, wie es Chamberlain et al. beschreiben: „While a holy grail of parallel computing has historically been to automatically transform good sequential codes into good parallel codes, our faith in compilers [...] does not extend this far given present-day technology“ [19, S. 292]. Nichtsdestotrotz gibt es unterstützende Compiler und Präprozessoren, etwa zur Erkennung von Synchronisationspunkten [27, 92]. Zur Kommunikation wird zumeist PGAS eingesetzt, wobei es eine Notation zwischen lokalen und fernen Daten gibt, beispielsweise globale Pointer (Split-C) oder eine spezielle Syntax (CAF). Viele dieser Lösungen wurden im Laufe der aufgegeben, doch einige wenige sind bis heute in Verwendung; entweder in ihrer ursprünglichen Form oder durch Weiterentwicklungen. Ein Beispiel dafür ist das Konzept von Co-Arrays in Fortran, das in der Wettervorhersage und im Deep Learning verwendet wird [68]. Ein Nachfolger von Cilk ist Cilk Plus [70], während UPC durch UPC++ wieder an Aufmerksamkeit gewonnen hat [3]. UPC++ ist eine moderne Bibliothek für C++, die dem Programmierer Datentypen zur Repräsentation von Tasks und verteiltem Speicher zur Verfügung stellt ohne neue Schlüsselwörter einzuführen und intern GASNet-EX zur Kommunikation verwendet.

3.2.3 Python-Ökosystem

Durch seine Einfachheit, Kürze und Klarheit ist Python für das Prototyping im wissenschaftlichen Rechnen beliebt [41]. Darüber hinaus ist Python ein Standardansatz im Bereich des Deep Learning durch Bibliotheken wie TensorFlow. Im Zentrum des wissenschaftlichen Rechnens mit Python steht die Bibliothek NumPy, die alle wichtigen Operationen mit Arrays abbildet und dafür auf hochperformanten Code in C oder Fortran zurückgreift [41]. Aus diesem Grund wird sie auch als ein Backend von autoBLAS unterstützt. NumPy arbeitet jedoch nicht verteilt, weder mit mehreren Threads noch mit mehreren Knoten eines Clusters [71]. Aus diesem Grund wurden mehrere Bibliotheken entwickelt, um Operationen mit NumPy zu verteilen, namentlich Dask [71], Theano [10], Phylanx [82], Spartan [43], Arkouda [60] und Legate [6]. Üblicherweise imitieren diese die NumPy-Schnittstelle, damit ein Einsatz in bereits mit NumPy geschriebenen Programmen möglich ist, indem die Importanweisung ausgetauscht wird. Ebenso ist eine Interoperabilität mit NumPy meistens gegeben. Auch darüber hinaus sind sich die Ansätze konzeptuell vergleichsweise ähnlich. Programme werden zumeist in Form von Taskgraphen dargestellt, die vor der Ausführung mehrere Verarbeitungsschritte durchlaufen, etwa Optimierungen oder Scheduling. Die Ausführung selbst übernimmt für die mit diesen Bibliotheken geschriebenen Programme im Wesentlichen nicht direkt der Pythoninterpreter. Einige der Lösungen nutzen dafür hochperformante Laufzeitumgebungen, auf die später noch genauer eingegangen wird. Im Falle von Phylanx ist dies HPX, bei Arkouda Chapel und bei Legate das Legion-System. Dask, Theano und Spartan setzen für die Ausführung auf jit- oder aot-kompilierten Code, teilweise auch für GPUs. Insofern wird Python lediglich als einfach zu nutzendes Frontend gewählt, um die komplexen und hochperformanten Backends zu steuern. Gemäß dieser Philosophie setzen diese Bibliotheken darauf, dem Nutzer so viele Entscheidungen wie möglich abzunehmen. So sind die Parallelisierung, das Scheduling und die Auszeichnung von Abhängigkeiten von Tasks und Datenbereichen nicht notwendig, sondern werden implizit umgesetzt.

3.2.4 Komplexe Softwaresysteme

Durch die Möglichkeiten, die moderne Methoden und Plattformen der Softwareentwicklung zusammen mit globaler Kollaboration bieten, sind in den letzten Jahren viele sehr komplexe Lösungen zum Verteilen ernormer Datenmengen entwickelt worden. Dazu zählen einerseits Big-Data-Plattformen auf Java-Basis wie Apache Spark und Apache Flink, die hauptsächlich zur Datenanalyse verwendet werden. Daneben gibt es auch Laufzeitumgebungen für das Hochleistungsrechnen im Wissenschaftsumfeld, die zumeist in C oder C++ geschrieben sind. Zu diesen zählen HPX [46], Chapel [19], Legion [7], PaRSEC [15] und X10 [22]. All diese Laufzeitsysteme laufen auf heterogenen Clustern mit verteiltem Speicher und wurden teilweise auf leistungsstarken Supercomputern getestet. Die Unterstützung von Hardwarebeschleunigern wie GPUs ist nicht flächendeckend vorhanden, lässt sich aber gegebenenfalls durch externe Bibliotheken nachrüsten, wie im Falle von Chapel [18]. Im Wesentlichen handelt es sich bei diesen Frameworks um asynchrone, parallele Taskverarbeitungssysteme. Ein zentraler Bestandteil ist also der Scheduler, der die Tasks den zur Verfügung stehenden Knoten zuweist, und häufig selbst verteilt wird. Ziele dieser Scheduler sind Datentransfers und Kommunikation zu minimieren, Wartezeiten mit Berechnungen zu überlagern und dabei die Eigenschaften der Knoten und des Netzwerks, etwa die GPU-Verfügbarkeit beziehungsweise die Bandbreite zu berücksichtigen [7]. Eine Leitlinie dafür ist, den Transfer der Tasks zu den Daten dem Transfer der Daten zu den Tasks vorzuziehen [45, S. 1]. Dabei ist es noch immer eine offene

Herausforderung „how to maximize the locality of access to array data spread out across the memory of many machines“ [43, S. 1]. Trotz dessen hat dynamische, asynchrone Ausführung von Tasks gegenüber einer statischen Ausführung wie bei MPI den Vorteil, dass Latenzen und Schwankungen zur Laufzeit besser mit Berechnungen überbrückt werden können, da hier mehr Informationen vorliegen [15, S. 38]. Für die Deklaration der Tasks gibt es kein einheitliches Muster, da diese Systeme sehr unterschiedlich angesprochen werden. Chapel und X10 definieren jeweils gleichnamige, eigenständige Programmiersprachen, während PaRSEC und HPX Schnittstellen für C beziehungsweise C++ definieren. Das Legion-System bietet sowohl Bindings für C++ als auch eine eigene Programmiersprache namens Regent [75] an. Zusätzlich lassen sich einige dieser Systeme durch Pythonbibliotheken nutzen, wie zuvor beschrieben wurde. Als Kommunikationsmodell wird überwiegend das PGAS-Modell verwendet. HPX nutzt eine Weiterentwicklung namens AGAS [45], während PaRSEC auf Nachrichtenübertragung setzt [81, S. 1431].

Komplexe Softwaresysteme neigen dazu, Schwierigkeiten beim Kompilieren und Linken zu verursachen. Das gilt insbesondere dann, wenn sie in C, C++ oder Fortran geschrieben sind, da hier umfassende Buildsysteme inklusive Abhängigkeitsverwaltung fehlen, die es in modernen Sprachen wie Rust (mit Cargo) oder Go gibt.

3.2.5 Zusammenfassung

In diesem Abschnitt wurde eine Auswahl an Sprachen, Spracherweiterungen und Frameworks für verteiltes Rechnen aus den letzten 40 Jahren vorgestellt. Für die weitere Betrachtung innerhalb dieser Arbeit wird an dieser Stelle eine Filterung vorgenommen. Zunächst einmal liegt der Fokus stark auf Bibliotheken, die sich aus C++ oder Python heraus nutzen lassen, um den Implementierungsaufwand für das Prototyping in autoBLAS gering zu halten. Eine neue Sprache in autoBLAS einzubinden bedeutet beispielsweise, dass viel Code für die bloße Unterstützung geschrieben werden muss, der noch nichts mit der eigentlichen Verteilung zu tun hat. Im Folgenden werden zudem nur noch Frameworks betrachtet, die heutigen Qualitätsansprüchen genügen und auch von der Ausrichtung her gut zu autoBLAS passen. Die Qualitätsansprüche für diese Arbeit umfassen einerseits, dass die Frameworks noch immer gepflegt werden müssen. Damit ist nicht zwingend gemeint, dass sie regelmäßig neue Funktionen bekommen müssen, jedoch sollte es noch Entwickler geben, die eventuell entdeckte Fehler oder Sicherheitslücken beheben. Dies schließt bereits eine Vielzahl an älteren Frameworks aus, die inzwischen beispielsweise von Folgeframeworks abgelöst wurden, wie dies etwa bei ZPL [20] und Chapel oder Sequoia [33] und Legion der Fall ist. Weiterhin wurden nur Frameworks in Betracht gezogen, die sich kostenlos nutzen lassen, also im besten Fall unter einer Open-Source-Lizenz stehen. Außerdem ist es wichtig, dass sich die Lösung mit einer beigefügten Anleitung kompilieren oder zumindest herunterladen und installieren lässt. Diese Bedingung ist insbesondere für viele Frameworks, die in C oder C++ geschrieben sind, nicht gegeben. Hier kommt es durch globale Präprozessormakros oder nicht dokumentierte Abhängigkeiten häufig zu Problemen beim Kompilieren und/oder Linken, die sich nicht ohne größeren Aufwand beheben lassen. Dadurch mussten auch ansonsten vielversprechende Kandidaten aussortiert werden, wie etwa PBLAS [24]. Zuletzt wurde von konzeptuell sehr ähnlichen Lösungen jeweils nur ein Vertreter ausgewählt. Auf diese Weise wurden für die detailliertere Analyse folgende Frameworks ausgewählt: MPI als bewährte Schnittstelle für die feingranulare und benutzergesteuerte Verteilung von Berechnungen und Legion als umfangreiches, taskbasiertes und noch immer gepflegtes System sowie Dask als Pythonbibliothek für maximalen Komfort.

3.3 Theoretische Analyse der verbleibenden Frameworks

Dieser Abschnitt befasst sich mit einer tiefergehenden theoretischen Analyse der bisher ausgewählten Frameworks. Die Ziele sind dabei denen des vorangegangenen Abschnitts ähnlich. Einerseits sollen Erkenntnisse aus den verschiedenen Ansätzen vorgestellt werden. Andererseits dient dies dem Feststellen eventueller Diskrepanzen zwischen autoBLAS' Ausrichtung und der der hier beschriebenen Frameworks.

3.3.1 MPI

Das *Message Passing Interface* (MPI) ist eine Schnittstelle für das verteilte Rechnen, die Anfang der Neunzigerjahre vorgestellt wurde [78]. Als Ziele dieser Schnittstelle wurden Portabilität, Effizienz und Einfachheit in der Benutzung angegeben [78, S. 878]. Ähnlich wie für die BLAS gibt es auch für MPI zahlreiche Implementierungen und Erweiterungen. Beispiele für bedeutende Implementierungen sind OpenMPI [34], MPICH[1] und MS-MPI [2]. Einen modernen Wrapper für C++ mit Objektorientierung bietet `Boost.MPI` [3].

In MPI wird jedem Knoten eine ID zugeordnet, die *Rank* genannt wird. Diese ID ist innerhalb einer Prozessgruppe einmalig [78, S. 879]. Weiterhin sind zahlreiche Funktionen definiert, mit denen Daten effizient mit anderen Knoten geteilt werden können, indem entweder deren Rank angesprochen wird, oder ein beliebiges Ziel. Die Kommunikation in MPI kann synchron oder asynchron ablaufen [78, S. 881]. Zusätzlich kann angegeben werden, ob die Funktionsaufrufe zum Senden und Empfangen blockieren sollen oder nicht. Bei den nichtblockierenden Varianten ist es möglich, die versendeten Daten zu korrumpieren, indem sie geändert werden, bevor der Versand vollständig abgeschlossen ist. Auf ähnliche Weise kann ungültiger Speicher gelesen werden, indem beim Sender darauf zugegriffen wird, bevor der Empfang abgeschlossen ist. Die blockierenden Varianten kehren erst aus dem Aufruf zurück, wenn die Wiederverwendung der involvierten Speicherbereiche sicher ist.

Eine große Stärke von MPI ist die koordinierte Kommunikation mehrerer Knoten einer Gruppe [78]. Diese umfasst sowohl die koordinierte Berechnung als auch das Verteilen von Daten. Beides ist von zentraler Bedeutung für diese Arbeit. Zum Verteilen definiert MPI drei Möglichkeiten mit jeweils zwei Modi, die im Folgenden kurz beschrieben werden. Nach dem *Broadcasting* haben alle Knoten dieselben Daten im Empfangspuffer. Eine Unterscheidung wird danach getroffen, wie viele Sender es

---

[1] `https://github.com/pmodels/mpich`

[2] `https://github.com/Microsoft/Microsoft-MPI`

[3] `https://www.boost.org/doc/libs/1_64_0/doc/html/mpi.html`

gibt. Neben dem *One-to-All-Broadcasting* gibt es das *All-to-All-Broadcasting*, bei dem jeder Knoten Daten an jeden anderen sendet und anschließend alle eingehenden Daten zusammenfügt. Diese Art der Kommunikation ist im autoBLAS-Kontext sinnvoll, um (kleinere) Daten auf jedem Knoten verfügbar zu machen, wie etwa Skalare oder Vektoren, die für mehrere Berechnungen benötigt werden. *Scattering* funktioniert ähnlich wie Broadcasting, allerdings erhält im Allgemeinen nicht jeder Empfänger dieselben Daten. Auch hier gibt es *One-to-All-* und *All-to-All-*Modi, die bestimmen, wie viele Sender beteiligt sind. Algorithmen für blockunterteilte Matrizen können mit Scattering effizient implementiert werden. Schließlich gibt es das *Gathering*, bei dem Knoten Daten von allen anderen Knoten anfragen. Insofern ist dies die inverse Operation zum Scattering und kann für ähnliche Zwecke verwendet werden. Die *All-to-All-*Varianten vom Gathering und Scattering erzeugen das gleiche Ergebnis, da jeweils jeder Knoten Daten an jeden anderen versendet.

Schließlich bietet MPI Funktionalitäten für *Reducing* und *Scanning*. Reducing ist im Wesentlichen eine Erweiterung des One-All-Gatherings, die die empfangenen Daten nicht konkateniert, sondern mithilfe einer reduzierenden Funktion (etwa Summe) kombiniert [79, S. 224]. Scanning ist eine Form des Reducings, bei dem nicht lediglich das Endergebnis zu einem Knoten übertragen wird. Stattdessen werden die Zwischenergebnisse der Schritte kleiner gleich i im Knoten mit dem Rang i gesammelt [79, S. 246].

Als großer Nachteil von MPI wird gesehen, dass „the burden of hiding long latency operations […] by overlapping computation and communication is placed directly on the programmer“ [5, S. 55]. In diesem Fall liegt die Last auf autoBLAS, was insbesondere in Kombination mit dem bereits angesprochenen Nichtentscheidbarkeitstheorem zu suboptimalen generierten Berechnungsschritten führen könnte.

3.3.2 Legion

Im Jahr 2012 haben Bauer et al. Legion vorgestellt [7]. Hierbei handelt es sich um ein Laufzeitsystem, das dynamisch und automatisch Möglichkeiten der Parallelisierung in Programmen umsetzt. Die Implementierung als Laufzeitsystem bringt einige Vorteile mit sich. Wichtige Entscheidungen zur Verteilung von Daten und Tasks können bis zu dem Zeitpunkt aufgeschoben werden, in dem die meisten Informationen vorliegen, was die Entscheidungsqualität steigern kann. Zu diesen Informationen zählen die tatsächliche Datenmenge sowie die Bandbreite und Speichergrößen der Knoten. Zur Nutzung des Systems müssen die Abhängigkeiten zwischen den einzelnen Tasks vom Programmierer deklariert werden. Dazu bietet Legion ihm mehrere Schnittstellen an. Es kann sowohl als Bibliothek für C++ genutzt werden, als auch mittels der eigenen Sprachen Regent [74, 75] und Core Legion [84]. Dadurch bietet Legion dem Benutzer alle Vorteile der Technologiekategorie nach Thoman et al., da er frei wählen kann, welche Schnittstelle am besten zu seinem aktuellen Anwendungsfall passt. Allerdings ist ein Wechsel zwischen den Schnittstellen nicht so einfach möglich, wie es etwa bei autoBLAS der Fall ist.

Legions Laufzeitsystem besteht aus mehreren Teilen, die voneinander verschiedene Aufgaben wahrnehmen (Abbildung 3.2). Dies ermöglicht es beispielsweise, dass sie vom Nutzer einzeln ausgetauscht werden können, um das System an dessen Be-

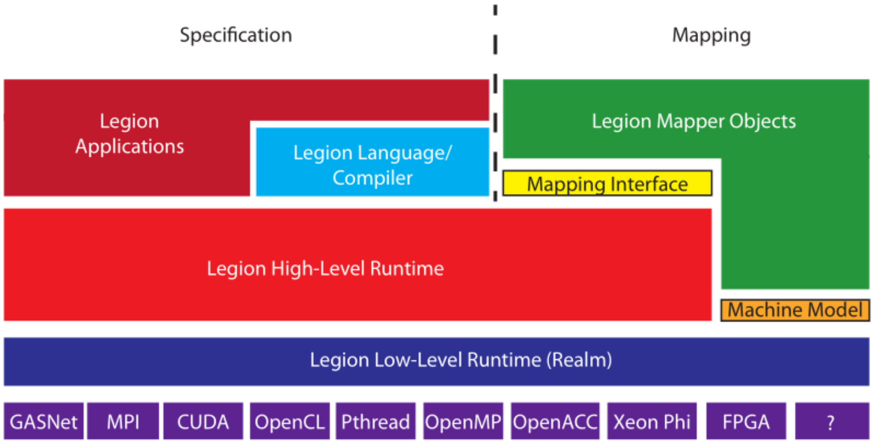

Abbildung 3.2: Übersicht der Legion-Architektur mit den verschiedenen Abstraktionsleveln und Aufgabenteilungen
Quelle: [5, S. 51]

dürfnisse anzupassen. Eine wichtige Komponente dieses Systems ist der sogenannte „Software out-of-order processor“ (SOOP[4]) [7]. Dieser hat als Teil der High-Level-Runtime die Aufgabe, die Daten und Tasks auf die Knoten zu verteilen, und arbeitet selbst verteilt, um beispielsweise rekursive Tasks handhaben zu können [7]. Zusätzlich wartet er nicht darauf, bis die Tasks abgeschlossen sind, sondern teilt unabhängig von ihrer Abarbeitung Daten und Tasks zu. Während seiner Arbeit durchläuft Legions SOOP mehrere Stufen.

In der ersten Stufe werden die Abhängigkeiten zwischen den einzelnen Tasks bestimmt. Ein Task ist von einem anderen Task abhängig, falls beide in konfligierender Weise auf dieselbe logische Region zugreifen. Ob ein Konflikt vorliegt, wird wiederum von Annotationen abgeleitet, die der Programmierer im Legion-Programm angibt. Beispielsweise erzeugen mehrere Lesezugriffe auf dieselbe Region keinen Konflikt. Wichtig für die Effizienz dieser ersten Stufe ist die Einschränkung von Legion, dass Kindtasks nur auf dieselben Regionen wie ihre Elterntasks zugreifen dürfen und dies auch maximal mit denselben Zugriffsrechten, denn dadurch wird Folgendes erreicht: „Let t_1 and t_2 be two sibling tasks with no dependence. Then no subtask of t_1 has a dependence with any subtask of t_2.“ [7, S. 4]. Dadurch muss die Abhängigkeitsanalyse nicht für jedes Paar von Tasks durchgeführt werden. Allerdings muss bei der Analyse berücksichtigt werden, dass Regionen partitioniert werden können. Dies bedeutet, dass eine logische Region, beispielsweise die Kanten eines Graphen, auf mehreren Clusterknoten verteilt beziehungsweise repliziert gespeichert sein kann. Damit können die tatsächlich für eine Berechnung benötigten Daten zielgenau angesprochen werden. Vorteile dessen sind zum einen weniger Datentransfers zwischen den Maschinen mit jeweils geringerem Umfang und andererseits weniger benötigter Platz auf Geräten mit begrenztem Speicher wie etwa GPUs. Durch die eventuelle Partitionierung sind insbesondere die Fälle zu beachten, in denen sich mehrere Subregionen

---

[4] In der Doktorarbeit von Bauer wird er stets nur „runtime“ genannt, aber die Analogie zu Out-of-order-Pipelines wird dennoch hergestellt [5, S. 58]

überlappen, das heißt sich gemeinsame Speicherbereiche teilen. Um den Analyseaufwand zu verringern, wird auch für die Regionen eine baumartige Datenstruktur mit Eltern-Kind-Beziehungen verwendet [7]. Dieses Konzept der Speicherhierarchie existiert auch bei Sequoia [33], aus dem heraus sich Legion entwickelt hat [7]. Dadurch ist es überflüssig zu prüfen, ob sich zwei Subregionen überlappen, wenn es keine gemeinsamen Vorfahren gibt, die sich überlappen. Nachdem die Abhängigkeiten zwischen den Tasks bestimmt wurden, geht es im nächsten Schritt um die tatsächliche Verteilung. Hier wartet ein Task, bis allen anderen Tasks, von denen er abhängt, Knoten zugewiesen und deren Daten verteilt wurden. Anschließend wird der besagte Task verteilt, indem er durch einen SOOP auf einem anderen Knoten gestohlen („pull") oder durch den SOOP auf dem aktuellen Knoten zu einem anderen Knoten geschickt („push") wird [7]. Im dritten Schritt werden die physischen Instanzen der Regionen bestimmt, die der Task verwendet. Dazu müssen zunächst diejenigen Knoten ausfindig gemacht werden, die eine aktuelle Version der Daten gespeichert haben, und davon einer für den Task ausgewählt werden. Die hierzu verwendeten Algorithmen sind ähnlich wie die für den ersten Schritt. Schließlich wird der betrachtete Task ausgeführt. Jegliche Kindtasks, die dabei aufgerufen werden, werden in den SOOP des Knotens geladen, der den Task ausführt, und durchlaufen den oben beschriebenen Schedulingvorgang. Zuletzt läuft ein Aufräumvorgang, der unter anderem nicht mehr genutzte Regionen freigibt [7].

Ein wichtiger Bestandteil von Legion ist das bereits erwähnte Mapping-Interface (Abbildung 3.2). Es kann dazu genutzt werden, das Verhalten des Legionsystems maßgeblich anzupassen. Legion bringt einen Defaultmapper mit, dessen Methoden durch eine eigene Implementierung überschrieben werden können. Die Mapper werden von den SOOPs aufgerufen und bieten unter anderem Methoden zur Entscheidung, ob eine Pull-Anfrage von anderen SOOPs erlaubt werden soll oder welche physische Instanz für eine logische Region verwendet werden soll [5, S. 149].

Ein Problem von Legion ist der hohe Kommunikationsaufwand [7]. Dieser ist einerseits aufgabeninhärent, andererseits wird er aber auch durch den Transfer von Tasks beziehungsweise den dazugehörigen Anfragen verursacht. Dieser durch Legion selbst verursachte Kommunikationsaufwand kann durch ein statisches Verteilen eingespart werden.

Legate ist eine Anbindung der für Datenwissenschaften zentralen Pythonbibliothek NumPy an das Legion-System [6]. Sie ist als ein Drop-In-Ersatz für NumPy konzipiert, was bedeutet, dass jeglicher mit NumPy geschriebener Code auch mit Legate korrekt funktioniert, indem die Importanweisung von `numpy` auf `legate.numpy` geändert wird. Dadurch geben alle Funktionen, die Arrays erzeugen, verteilt arbeitende `LegateArrays` zurück. Damit arbeitende Algorithmen profitieren in der Folge von der automatischen und transparenten Verteilung im Legion-System. Dadurch müssen aufgerufene Bibliotheksfunktionen nicht einmal zwingend selbst Legate nutzen. Sogar, wenn besagte Funktionen selbst NumPy-Arrays erzeugen und zurückgeben, ist dies kein Problem, da diese von Legate erfasst und konvertiert werden – oft ohne, dass Kopien notwendig sind [6]. Insgesamt macht es Legate damit einfach, Daten verteilt zu laden und bestehende Algorithmen darauf anzuwenden.

Intern verwendet Legate logische Regionen für NumPys Arrays. Genauer wird eine logische Region für jeden Arrayshape angelegt [6]. Dies vereinfacht die Partitionierung von Arrays mit demselben Shape, wenn beide miteinander verrechnet werden sollen (zum Beispiel bei elementweisen Operationen). Die Funktionen und Methoden der NumPy-API werden in Legate mittels Futures abgebildet, was bedeutet, dass sie sofort zurückkehren und erst blockieren, wenn versucht wird, Ergebnisse zu lesen, die noch nicht berechnet wurden. Wichtig zu erwähnen ist hierbei, dass Legate zur eigentlichen Berechnung ebenfalls BLAS-Versionen nutzt, etwa OpenBLAS [90] oder cuBLAS [6].

Letztendlich wird Legate aufgrund seines schwierigen Setups zugunsten von Dask nicht in den Experimenten verwendet. Auch wenn es eine Pythonbibliothek ist, muss die gesamte Legion-Laufzeitumgebung mitinstalliert werden. Es werden zwar offizielle Dockerimages angeboten, doch diese sind mit 31 GB sehr schwergewichtig und zudem an die Verfügbarkeit bestimmter Grafikkarten gebunden.

3.3.3 Dask

Die Pythonbibliothek Dask lässt sich einfach über den Paketmanager pip installieren. Auch in der Anwendung ist sie benutzerfreundlich, da sie die NumPy-Schnittstelle imitiert. Im Gegensatz zu NumPy führt sie die Berechnungen jedoch träge durch, das heißt wenn auf Dask-Arrays Operationen angewendet werden, werden sie nicht direkt ausgeführt, sondern zunächst zu einem Graphen zusammengesetzt [71]. Erst, wenn ein Ergebnis benötigt wird, werden Optimierungen oder Kompilationen vorgenommen. Aus diesem Grund ist Dask nur bedingt für die Ausführung von Berechnungen in Schleifen geeignet, da in diesem Fall zunächst ein umfangreicher Graph aufgebaut wird, dessen Verarbeitung beim Abrufen des Ergebnisses viel Zeit in Anspruch nimmt, bevor die eigentliche Berechnung erfolgt. Insbesondere für kleinere Datenmengen ist Dask in dieser Situation deshalb deutlich langsamer als NumPy selbst. Dies ist jedoch nicht direkt ein Problem für autoBLAS, da es innerhalb der Blöcke keine Schleifen gibt und die Schnittstelle nach außen wieder durch NumPy-Arrays abgebildet wird. Die Integration von Dask in autoBLAS gestaltet sich konzeptuell sehr einfach. Es gibt keine expliziten Kommunikationsaufrufe, die durch autoBLAS eingefügt werden müssen. Des Weiteren muss innerhalb eines Programms keine Funktion zum Initialisieren oder Finalisieren aufgerufen werden. Es müssen lediglich Konvertierungen zwischen NumPy- und Dask-Arrays zu Beginn und Ende eines Blockes vorgenommen werden, um die Transparenz für den Anwender zu wahren.

4 Verteilung von Ausdrücken linearer Algebra

In diesem Kapitel wird darauf eingegangen, wie sich Ausdrücke der linearen Algebra auf die Knoten eines Hochleistungsclusters verteilen lassen. Zunächst wird dies auf einer konzeptuellen und frameworkunabhängigen Ebene besprochen. Insbesondere ist es hier bereits möglich, Optimierungen zu identifizieren, oder etwa, welche Operationen sich trivial verteilen lassen. Anschließend werden konkrete Implementierungen mithilfe der zuvor ausgewählten Frameworks vorgestellt. Dabei liegt der Fokus nicht darauf, wie ein menschlicher Programmierer die Verteilungen mit dem jeweiligen Framework durch seine Intuition möglichst optimal implementieren würde, sondern vielmehr darauf, wie die Verteilung mit autoBLAS' algorithmischen Möglichkeiten umgesetzt werden könnte.

4.1 Allgemeine Überlegungen

Sinn der Verteilung Eine Verteilung im Hochleistungsrechnen hat im Allgemeinen die Absicht, Berechnungen zu beschleunigen beziehungsweise überhaupt erst zu ermöglichen. Dem gegenüber steht ein bei steigendem Verteilungsgrad wachsender Synchronisations- und Kommunikationsaufwand. Während CPU-lastige Arbeit effizient durchgeführt werden kann, benötigt die Kommunikation über ein Netzwerk vergleichsweise lange Zeit. Zusätzlich beeinflussen gegebenenfalls bereits Designentscheidungen beim Entwurf der Algorithmen und Datenstrukturen, wie effizient ein Programm auf einem Kern und wie effizient es in einer verteilten Umgebung laufen kann. Oft kann ein einzelner CPU-Kern eine Berechnung schneller erledigen als ein Cluster mit vielen Knoten, wenn man die Methoden des Algorithm Engineering richtig anwendet [58]. Aus diesem Grund muss immer abgewägt werden, ob eine Verteilung überhaupt sinnvoll ist. Dies ist etwa abhängig vom Umfang der zu verarbeitenden Daten oder der Parallelisierbarkeit der involvierten Operationen. In dieser Arbeit wird davon ausgegangen, dass ausreichende Gründe gegeben sind, die für eine Verteilung sprechen. Das heißt, dass autoBLAS nicht den Versuch unternehmen wird zu schätzen, welchen Gewinn eine Verteilung der Ausdrücke in einem vorliegenden Block haben wird, und basierend darauf Entscheidungen zu treffen. Nachdem nun geklärt wurde, in welchen Fällen sich eine Verteilung vor allem lohnt, wird mit damit fortgefahren, wie eine Verteilung umgesetzt werden kann.

Arten der Verteilung Gegebene Ausdrücke können entlang verschiedener, zueinander orthogonaler Achsen verteilt werden. Zunächst können voneinander unabhängige Teilausdrücke von verschiedenen Knoten bearbeitet werden. Diese Art ist konzeptuell einfach umzusetzen und generell anwendbar, wenn die Operanden und Ergebnisse der Teilausdrücke in den Speicher der jeweiligen Knoten passen. Die Effektivität sinkt jedoch, wenn die Berechnung größtenteils linear erfolgt, das heißt, wenn jeder Teilausdruck von einem Großteil der vorherigen abhängig ist. Dann verbringen die Knoten unter Umständen einen signifikaten Teil der Laufzeit mit Warten. In diesem Fall bietet es sich an, die Berechnung der einzelnen Operationen auf verschiedene Knoten zu verteilen, was die zweite der oben erwähnten Achsen darstellt. Dies ist nicht immer sinnvoll möglich, wie die folgende Unterteilung aufzeigt. Auf der obersten Ebene gibt es *unabhängig* und *abhängig* verteilbare Operationen. Zur ersten Gruppe gehören beispielsweise alle elementweisen Operationen, wie die skalare Multiplikation oder die Matrixaddition. Hier könnten beispielsweise die oberen und unteren Teile zweier Matrizen getrennt voneinander addiert werden. Die abhängigen Operationen werden weiter unterteilt in *lokal abhängige* und *global abhängige.* Die global abhängigen Operatoren sind im Allgemeinen nicht gut auf dieser Ebene verteilbar. Zu ihnen zählt beispielsweise die exakte Bestimmung der Determinante einer beliebigen, quadratischen Matrix, welche ohne weitere Metainformationen die gesamte Matrix braucht. Lokal abhängige Operationen hingegen benötigen zur Erzeugung des Endergebnisses zwar die gesamten Argumente, sind aber im Sinne von Devide and Conquer in unabhängige Teilprobleme zerlegbar. Ein Beispiel hierfür ist `sum`, bei dem etwa der obere und untere Teil einer Matrix auf verschiedenen Knoten summiert und die Endergebnisse anschließend zusammengeführt werden können. Auch das wichtige Standardskalarprodukt zählt in diese Kategorie, denn es besteht aus einer (unabhängigen) elementweisen Multiplikation und einer Summierung.

Die Minimierung von Kommunikation und Datenübertragung ist eines der wichtigsten Ziele bei der Verteilung von Berechnungen [8]. Dies ist häufig noch wichtiger als den Umfang der Daten klein zu halten, da Hochleistungsrechner oft große Netzwerkbandbreiten mitbringen, aber durch die Latenzen dennoch unvermeidbare Wartezeiten entstehen, sobald Netzwerkaufrufe stattfinden. Dies hat zur Folge, dass Zwischenergebnisse erst übertragen werden sollten, wenn sie tatsächlich auf einem anderen Knoten benötigt werden. Zusätzlich sollten so viele Operationen wie möglich angewendet werden, solange die Daten verteilt sind. Insbesondere sollten reduzierende Operationen angewandt werden, bevor Daten versendet werden, um die zu übermittelnde Datenmenge zu minimieren. Gleiches gilt für die Berechnungsreihenfolge von Matrixmultiplikationen. Diese sollten im Rahmen des Assoziativgesetzes so sortiert werden, dass die Ergebnisse eine minimale Dimension aufweisen.

Semantikunterschiede beim verteilten Rechnen Viele Modelle des verteilten Rechnens stützen sich auf das Kopieren von Daten zwischen verschiedenen Knoten. Durch diese Kopien kann es zu semantischen Unterschieden zwischen dem bisherigen Fall mit einem Knoten und einem verteilten Fall kommen. Ein Auslöser dafür ist Aliasing, das heißt, wenn sich mehrere Variablen (unbeabsichtigt) einen Speicherbereich teilen. Die auftretenden Semantikunterschiede basieren darauf, dass bei einer Ausführung durch einen Knoten mit einem Speicher die Variablen stets gleichzeitig aktualisiert werden, während in einer Mehrknotenausführung durch das Kopieren alle Variablen unabhängig voneinander geändert werden können. In der Folge können Wettlaufbedingungen beim Zurückschreiben und unerwartete Ergebnisse bei

Listing 4.1: Beispiel für einen Semantikunterschied zwischen Ein- und Mehrknotenausführung durch Aliasing

```
1 #autoblas {
2   Vector a data=ptr;
3   Vector b data=ptr;
4   Vector c;
5   b = 2*b;
6   c = a - b;
7 }
```

Folgerechnungen auftreten. In Listing 4.1 ist ein Beispiel dafür aufgeführt. Eine Ausführung auf nur einem Knoten wird in Zeile 5 sowohl `a` als auch `b` ändern, da ihr `data`-Metadatum auf denselben Speicherbereich verweist. Werden beide Variablen allerdings für die Berechnungen auf andere Knoten kopiert, kann es dazu kommen, dass `c` am Ende des Blockes nicht gleich dem Nullvektor, sondern `-a` ist. Dieses Beispiel ist relativ übersichtlich, jedoch sind weit komplexere und subtilere Folgen möglich. Aus diesem Grund erfordert es autoBLAS folgend, dass die Speicherbereiche der Variablen innerhalb eines autoBLAS-Blocks disjunkt sind.

Besonderheiten im SPMD-Modell Im Wesentlichen geht es beim SPMD-Modell darum, dass dasselbe Programm auf mehreren Knoten simultan gestartet wird. Bis es zu Fallunterscheidungen kommt, die auf Daten basieren, die auf allen Knoten unterschiedlich sind, führen alle Knoten also auch denselben Code aus. Da autoBLAS seinen Nutzern transparent eine Performancesteigerung bieten möchte, enthalten Programme mit autoBLAS-Blöcken für gewöhnlich keine derartigen Fallunterscheidungen. Zum Schreibzeitpunkt sollte sich der Programmierer idealerweise keine Gedanken darüber machen müssen, ob sein Programm lokal mit einem Thread, lokal mit mehreren Threads beziehungsweise Prozessen oder verteilt auf einem Cluster laufen wird. Dadurch wird auf allen Knoten potenziell viel Code redundant ausgeführt, was zu einer Verschwendung von Ressourcen führt. Besonders gravierend ist dies beispielsweise beim Laden der zu verarbeitenden Daten. Wird das Programm zunächst sequentiell geschrieben, führt dies bei einer Ausführung im SPMD-Modell dazu, dass die Daten in jeden Knoten geladen werden und dort Arbeitsspeicher verbrauchen. Die Verteilungsfunktionalität von autoBLAS muss nicht zwingend mit dem SPMD-Modell umgesetzt werden, allerdings ist MPI, das diesem Modell entspricht, einer der Kandidaten, die zu verwenden am sinnvollsten scheint. Eine mögliche Lösung dafür ist, das Laden von Daten per Kommandozeilenargumente zu steuern.

autoBLAS' lokale Sicht Wie in Kapitel 2.3 bereits angedeutet wurde, ist das Verteilen von Berechnungen ein komplexer Prozess, bei dem viele Informationen integriert werden müssen. Dazu gehören etwa Informationen über den Aufruf- und Abhängigkeitsgraphen der Tasks und daraus abgeleitet etwa Aufrufhäufigkeiten. Zu vielen dieser Informationen hat autoBLAS jedoch keinen oder nur einen sehr beschränkten Zugang, da autoBLAS nur lokale Informationen von innerhalb der Blöcke zur Verfügung hat. Zum aktuellen Zeitpunkt kennt die Referenzimplementierung von autoBLAS das Konzept von Dateien nicht; es wird lediglich vom Standardinput gelesen und zum Standardoutput geschrieben. Aufgrund dessen kann autoBLAS keine Bezüge zwischen den Blöcken herstellen, um etwa Parallelisierungsmöglichkeiten zwischen ihnen festzustellen. Außerdem kann es nicht feststellen, ob ein Block etwa innerhalb einer Schleife aufgerufen wird, um die Schleifendurchläufe zu parallelisieren, falls dies möglich wäre. Weiterhin sollte autoBLAS es aus diesem Grund vermeiden, Konstrukte zu erzeugen, die einen großen Initialisierungsaufwand mit sich bringen. Anderenfalls könnte der eigentlich als Hochgeschwindigkeitscode angedachte Block zum Flaschenhals werden, wenn er häufig ausgeführt wird. Zumindest im Falle von Schleifen wäre es verhältnismäßig einfach, autoBLAS weitere Informationen durch den Programmierer zur Verfügung zu stellen, etwa in Form eines `#autoblas-for-` oder `#autoblas-while-`Konstrukts. In diesem Fall könnten aufwändige Initialisierungen nur einmalig vor der Schleife durchgeführt werden statt in jeder Iteration. Bisher ist autoBLAS jedoch am effektivsten, wenn die Blöcke groß sind, das heißt viele Berechnungen enthalten, und nicht in performancekritischen Schleifen aufgerufen werden.

autoBLAS-Header In vielerlei Hinsicht wäre es wünschenswert, wenn autoBLAS globale Informationen sammeln und bündeln, und selbst globale Objekte im Quellcode definieren könnte. Im bisherigen Modell ist dies allerdings unmöglich, da autoBLAS lediglich auf Code innerhalb der gekennzeichneten Regionen Lese- und Schreibrechte hat. Eine nützliche Erweiterung des Modells wäre daher etwa die Einführung einer von autoBLAS generierten Zusatzdatei. In C++ ließe sich dies beispielsweise durch einen Header abbilden, in Python durch ein Modul. In dieser Zusatzdatei könnte autoBLAS beliebige Objekte definieren und diese aus den verarbeiteten Dateien heraus benutzen. Dieses Vorgehen wäre mit einigen Vorteilen verbunden.

Zunächst einmal wäre dadurch die Generierung von übersichtlicherem Code als bisher möglich. Im aktuellen Modell ist autoBLAS nicht in der Lage, den generierten Code in mehrere, kleine Funktionen zu zerlegen – wenn ein Block viele Anweisungen enthält, wird er auch in einen großen Block mit viel unübersichtlichem Code umgewandelt. Dies entspricht nicht der guten Praxis der Softwareentwicklung [56, S. 31ff.]. Nachteile sind etwa eine erschwerte Lesbarkeit, Editierbarkeit und Wiederverwendbarkeit des Codes. Dies mag im Allgemeinen kein schwerwiegendes Problem sein, denn immerhin ist es ein Ziel von autoBLAS, dem Nutzer die Handhabung des konkreten Programmcodes abzunehmen. Dennoch kann es aus Gründen des Debuggings, der Sicherheit oder des Customizings sinnvoll sein, den generierten Code möglichst sauber und lesbar zu gestalten. Aus Sicht der Performance ist eine Feingliederung in mehrere Funktionen zumindest bei Backends für C++ unkritisch, da kleine Funktionen leicht für Compiler zu inlinen sind.

Eine zusätzlich generierte Datei hätte darüber hinaus den Vorteil, dass sie der globalen Sammlung und Aufbereitung von Informationen dienen kann. Eine Limitation dessen ist, dass es mit diesem Ansatz dennoch nicht möglich ist festzustellen, ob zwei autoBLAS-Blöcke parallel zueinander ausgeführt werden können, da nicht klar ist, ob beide auf denselben Speicherbereich zugreifen. Weiterhin kann daraus nicht abgeleitet werden, ob die autoBLAS-Blöcke jemals aufgerufen werden und falls ja, wie oft und in welcher Reihenfolge.

Verwandt mit dem vorherigen Punkt ist der Vorteil, dass eine Zusatzdatei Zustände speichern kann. Dies ist insbesondere dann wichtig, wenn Bibliotheken eine Initialisierungsphase vor der ersten Benutzung erfordern. In diesem Fall wäre es aus Performancesicht suboptimal, die Initialisierung in jedem einzelnen Block erneut vorzunehmen. Sinnvoll ist es stattdessen, eine Referenz auf ein initialisiertes Objekt der Bibliothek zu speichern und von den verschiedenen Blöcken darauf zuzugreifen. Auch eine Funktion zum Freigeben von Ressourcen kann hier untergebracht werden, etwa in Form eines Destruktors in C++ oder Exit-Hooks in anderen Sprachen[1].

Eine Zusatzdatei zu generieren würde allerdings eine bedeutende Änderung im aktuellen Arbeitsablauf mit der Referenzimplementierung hervorrufen. Bisher läuft jegliche Kommunikation mit dem Programm über Standardein- und -ausgabe, meistens für eine Eingabedatei nach der anderen. Bei einer Generierung von Hilfsdateien hätten autoBLAS-Aufrufe Nebeneffekte, die es zuvor nicht gab, was Skripte von Nutzern stören könnte. Da die autoBLAS-Referenzimplementierung vermutlich kaum Nutzer haben dürfte, wäre dies ein eher geringes Problem. Hinzu kommt ein konzeptuelles Problem: Um eine gemeinsame Hilfsdatei für alle autoBLAS-Blöcke anzulegen, wäre es notwendig, alle Quellcodedateien auf einmal einzulesen, was die Flexibilität bei der Arbeit mit autoBLAS einschränkt.

4.2 Zielsetzung

autoBLAS wurde mit Blick auf folgende Eigenschaften implementiert: Unabhängigkeit von der Hostsprache, Effizienz und Einfachheit. Entsprechend der Taxonomie von Thoman et al. werden in diesem Abschnitt die geplanten Merkmale für die Schnittstelle und Laufzeitumgebung von autoBLAS-Erweiterung erläutert, die sich stets an den genannten Zielen orientieren [81]. autoBLAS ist eine hostsprachenunabhängige Spracherweiterung, die im Wesentlichen aus einer Textersetzung besteht. Diese Unabhängigkeit wäre mit einer Bibliothek nicht in der vorliegenden Güte und Einfachheit realisierbar gewesen, trotz üblicher Schnittstellengeneratoren wie SWIG [9]. Auch die Benutzbarkeit einer solchen Bibliothek wäre sehr hostsprachenspezifisch, und es wären Kenntnisse in C++ erforderlich, um Zugriff auf die maximale Performance zu erhalten. Zusätzlich würde die Klarheit der Ausdrücke darunter leiden. Zuletzt spricht die Wiederverwendbarkeit beziehungsweise die Portabilität gegen eine Implementierung als Bibliothek: In der jetzigen Form ist es ohne weitere Probleme möglich, einen bestehenden autoBLAS-Block von einem Programm in ein anderes zu kopieren und ihn direkt zu nutzen. Gegen die Implementierung von autoBLAS als komplett eigene Sprache sprechen im Wesentlichen die in Abschnitt 3.1 genannten Gründe des erforderlichen Mehraufwands. Nichtsdestotrotz enthält autoBLAS typische Komponenten einer Programmiersprachenimplementierung wie Lexer, Parser, Optimierer und Codegeneratoren, wenn auch in vereinfachter Form verglichen mit einer vollständigen Programmiersprache. Weitere Details über die Softwarearchitektur der Referenzimplementierung sind im späteren Teil in Kapitel 6.1 aufgeführt. Schließlich bietet autoBLAS seinen Nutzern gegenüber

---

[1] etwa das Modul `atexit` der Python-Standardbibliothek

Bibliotheken oder sogar Sprachen den Vorteil, ein geringes Risiko bezüglich eines Lock-Ins darzustellen. Möchte man eine aktuell verwendete Bibliothek nicht mehr nutzen, müssen mitunter viele Quellcodedateien bearbeitet werden, was hohe Kosten verursachen kann. Wenn autoBLAS aus einem Projekt entfernt werden soll, muss lediglich einmalig Code für ein Backend der Wahl generiert werden, wodurch alle autoBLAS-Blöcke in anschließend manuell editierbaren Code umgewandelt werden.

Das Kommunikationsmodell, das autoBLAS zu seinen Nutzern hin bietet, ist ein Shared-Memory-Modell. Innerhalb der autoBLAS-Blöcke kann nicht angegeben werden, wo die zu den Eingabevariablen gehörenden Daten physisch gespeichert werden. Der Programmierer muss sich nicht um dieses Detail kümmern, sondern kann sich auf den zu implementierenden Algorithmus konzentrieren. Auch viele der weiteren Merkmale der Taxonomie von Thoman et al. sind bei autoBLAS implizit implementiert, um die Einfachheit und Klarheit der Sprache zu maximieren. Dem Benutzer werden damit viele Entscheidungen über Details abgenommen, die nicht wesentlich für die Lösung des eigentlichen Problems sind. Konkret heißt dies, dass der Speicher implizit verteilt wird, indem etwa das Verschieben oder Kopieren von Daten erfolgt, ohne dass der Nutzer einen wesentlichen Einfluss darauf hat. Dies gilt unter anderem für das Workermanagement, das Workmapping und die Synchronisation der Worker untereinander. Ob autoBLAS mit Hardwareheterogenität umgehen kann, hängt wiederum vom verwendeten Backend ab.

4.3 MPI

In diesem Abschnitt wird untersucht, wie autoBLAS die Verteilung mittels MPI implementieren könnte. Zu diesem Zweck wird zunächst für jede von autoBLAS unterstützte Funktion eine manuelle Verteilung unternommen und diese hinsichtlich ihrer Eigenschaften wie Effizienz und Korrektheitsüberlegungen betrachtet.

Allgemeine Überlegungen MPI-Programme benötigen Anweisungen für die Initialisierung ihrer Laufzeitumgebung, bevor MPI-Funktionen aufgerufen werden können. Zusätzlich wird empfohlen, die von MPI belegten Ressourcen am Ende der Nutzung wieder freizugeben. autoBLAS kann einerseits nicht ohne Weiteres annehmen, dass die MPI-Umgebung vor dem ersten autoBLAS-Block initialisiert wurde. Dies vom Nutzer zu verlangen, würde der Portabilität von autoBLAS-Programmen schaden, da das Verteilungsbackend dann nicht mehr einfach ausgewechselt werden könnte. Andererseits ist es für autoBLAS aber auch nicht möglich, die zur Initialisierung und Finalisierung benötigten Anweisungen für jeden autoBLAS-Block zu generieren, da dies von MPI verboten wird [79, S. 499, Z. 23].

Einen ersten Lösungsversuch stellt die Nutzung einer Präambel zu Beginn eines jeden autoBLAS-Blocks dar, die in etwa folgendermaßen aussehen könnte:

```
int initialized, finalized;
MPI_Initialized(&initialized);
MPI_Finalized(&finalized);
if (!initialized && !finalized) MPI_Init(nullptr, nullptr);
```

Damit ist jedoch nur das halbe Problem gelöst, denn autoBLAS kann nicht entscheiden, wann `MPI_Finalize` aufgerufen werden muss. Weiterhin könnte es mit solch einem Vorgehen zu Komplikationen kommen, wenn der Programmierer selbst MPI nutzt. Da autoBLAS transparent arbeitet, erwarten Nutzer womöglich nicht, dass MPI bereits durch eine andere Komponente (in dem Fall den autoBLAS-generierten Code) initialisiert wurde und setzen keine Abfrage vor die Initialisierung.

Einen zweiten Lösungsansatz hierfür können die in MPI 4.0 eingeführten Sessions bieten, allerdings wurde dieser Standard erst während der Schreibzeit dieser Arbeit veröffentlicht [79]. Deshalb ist er noch nicht weitläufig umgesetzt, wobei MPICH bereits eine Alphaversion bietet. Konkret wurde für die Experimente dieser Arbeit die Version 4.0a2 verwendet [2]. Ein Beispiel dafür, wie Sessions initialisiert werden können, ist im Standard zu finden [79, S. 507]. Die Effektivität dieses Vorgehens mit Sessions hängt von der Anzahl und Größe der autoBLAS-Blöcke ab. Am besten ist es geeignet, wenn es lediglich einen großen Block gibt, in dem alle intensiven Berechnungen durchgeführt werden.

MPI erlaubt die Definition verschiedener Gruppen, denen Knoten zugeteilt werden können. Damit ist es beispielsweise möglich einzuschränken, welche Knoten an einer koordinierten Kommunikation mittels Broadcast, Scatter und Gather teilnehmen. Diese Funktion wird hier nicht verwendet. Der Einfachheit halber werden alle Knoten derselben Gruppe zugeteilt. Praktisch könnte diese Aufteilung in autoBLAS verwendet werden, um die zur Verfügung stehenden Knoten entsprechend ihrer Kapazitäten und Ausstattung zu gliedern. Beispielsweise könnten Operationen, die viele Daten verarbeiten, einer Gruppe aus GPU-Knoten zugeteilt werden, oder einer mit Maschinen, die eine hohe Bandbreite haben. Auch die Latenz zwischen Knoten kann herangezogen werden, um die Knoten in Gruppen zu unterteilen, etwa wenn einige Knoten häufig miteinander kommunizieren müssen. Schließlich können Gruppen auch die Anzahl an Knoten einschränken, die an einer bestimmten Berechnung teilnehmen. Dies könnte erforderlich sein, um zu verhindern, dass jeder Knoten eine zu geringe Menge an Daten zugeteilt bekommt, sodass der Kommunikationsaufwand den Gewinn der Verteilung übersteigt.

Im Weiteren wird angenommen, dass alle Daten initial als Arrays des Typs `double` auf dem Knoten mit dem Rank 0 vorliegen. Zwar gibt es in MPI Funktionen zum verteilten Laden von Datensätzen [79, S. 643ff.], jedoch werden diese hier nicht weiter betrachtet. autoBLAS ist für das Verrechnen von Daten zuständig, bietet jedoch keine Funktion für das Laden von Daten. Dieser Teil ist größtenteils unabhängig vom verwendeten Backend und sollte deshalb außerhalb von autoBLAS-Blöcken stattfinden. Daher müssen die Daten vom generierten MPI-Programm nachträglich verteilt werden.

Insgesamt ist es dabei wichtig, die Daten so wenig wie möglich zu kopieren oder zu bewegen. Konkret heißt dies, dass Operationen so sortiert werden sollten, dass möglichst viele davon auf dieselben Daten eines Knotens angewendet werden können.

Unabhängige Operationen Die von autoBLAS unterstützten unabhängigen Operationen umfassen die Addition und Subtraktion, punktweise Multiplikation und Division sowie die punktweisen Exponential- und Logarithmusfunktionen für Skalare, Vektoren und Matrizen. Zusätzlich sind die skalare Multiplikation und Division von Vektoren und Matrizen definiert. Für diese Art von Operationen bietet sich die

---

[2] `https://github.com/pmodels/mpich/tree/581a631f375`

Scatter-Routine an. Mit ihr ist es möglich, die Vektor- und Matrix-Operanden aufzuteilen und die Stücke an alle teilnehmenden Knoten zu senden. Dabei ist es sinnvoll, die Operanden in kontinuierliche Speicherbereiche zu zerlegen. Die Folge daraus sind ein cache-optimiertes Auslesen sowie ein leichteres Zusammenfügen der Ergebnisse mit der Gather-Routine, die die empfangenen Daten lediglich konkateniert.

Ein Beispiel dafür, wie diese Art der verteilten Berechnung von autoBLAS in Code umgesetzt werden könnte, ist in Listing 4.3 skizziert. Es handelt sich hierbei noch nicht um eine Ausgabe von autoBLAS, sondern um eine vereinfachte Darstellung zur Veranschaulichung der Idee. Leicht zu erkennen in dieser Funktion sind die Aufrufe von `MPI_Scatter` und `MPI_Gather`, mit denen die Daten zunächst vom Root-Knoten an alle anderen gesendet werden und nach der Berechnung wieder zusammengesammelt werden. In der Praxis müssten die verallgemeinerten Routinen `MPI_Scatterv` [79, S. 208] und `MPI_Gatherv` [79, S. 198] verwendet werden, die es erlauben, für jeden Kommunikationsteilnehmer die Anzahl der zu sendenen beziehungsweise zu empfangenen Elemente zu spezifizieren. Allerdings sind für diese Routinen noch weitere Argumente definiert, etwa Offsets für überlappende oder nicht kontinuierlich im Speicher liegende Daten, welche für dieses Beispiel nicht relevant sind und damit die Klarheit vermindern würden. Aus diesem Grund wird hier angenommen, dass die Länge der Eingabearrays durch die Anzahl der Knoten teilbar ist. Die eigentliche Berechnung wird hier durch den `add`-Aufruf abgebildet, es ist jedoch jede Form der Berechnung möglich. Insbesondere kann hier ein weiterer autoBLAS-Block eingefügt werden, der in einem zweiten Durchgang durch eine konkrete BLAS-Implementierung ersetzt wird. Dies wird dadurch ermöglicht, dass die Menge der Daten autoBLAS ohnehin nicht bekannt ist und jeder Knoten ein eigenes autoBLAS-Programm berechnen kann, das sich lediglich in der Datenmenge von der Berechnung auf einem einzigen Knoten unterscheidet. Damit ist es leicht, von autoBLAS' Fähigkeiten in vollständigem Umfang Gebrauch zu machen, etwa der Einführung von temporären Variablen. Werden mehrere unabhängige Operationen hintereinander angewendet, müssen alle benötigten Daten verteilt und lokal verrechnet werden. Sind Skalare erst zur Laufzeit bekannt, bietet sich für deren Verteilung die `MPI_Bcast`-Funktion an, da damit eine Zahl effizient auf alle Knoten kopiert werden kann, ohne dass dafür ein Array angelegt werden muss, wie es mit `MPI_Scatter` der Fall wäre. Zur Kompilierzeit bekannte Skalare können natürlich direkt in das Programm kompiliert werden und verursachen damit keinen Kommunikationsaufwand. Erwähnenswert ist weiterhin die generische Definition der Funktion mittels eines beliebigen `MPI_Comm`, sodass sie auch im Sessionmodell verwendet werden kann.

Lokal abhängige Operationen In diese Kategorie fallen einige der wichtigsten Operationen überhaupt: die Produkte zwischen zwei Vektoren und/oder Matrizen. Zunächst wird sich der Vektor-Vektor-Multiplikation gewidmet. Listing 4.5 zeigt wieder eine Beispielimplementierung. Der Teil bis zur eigentlichen Berechnung ist analog zum vorherigen Beispiel 4.3. Naiv könnte das Skalarprodukt durch eine elementweise Multiplikation, gefolgt vom einem Gather und einer Summierung im Root-Knoten implementiert werden. Dies hätte jedoch einige Nachteile. Der offensichtliche davon ist, dass mehr Daten während des Gathers übertragen werden müssen als notwendig. Dadurch kann die Anwendung anfällig für Bandbreitenbegrenzungen werden. Indem

Listing 4.2: Definition einer Vektoraddition in autoBLAS

```
    Vector a;
    Vector b;
    Vector c;
    c = a + b;
```

Listing 4.3: mögliche Ausgabe von autoBLAS

```
void distributed_add(const double* a,
                     const double* b,
                     double* c,
                     const int length,
                     MPI_Comm comm)
{
  constexpr int ROOT_RANK = 0;
  int group_size;
  MPI_Comm_size(comm, &group_size);
  int rank;
  MPI_Comm_rank(comm, &rank);
  const int num_doubles_per_node = length / group_size;

  const double* operand1 = rank == ROOT_RANK ? a : nullptr;
  const double* operand2 = rank == ROOT_RANK ? b : nullptr;
  std::vector<double> output(num_doubles_per_node);

  std::vector<double> recv_buf1(num_doubles_per_node);
  std::vector<double> recv_buf2(num_doubles_per_node);
  MPI_Scatter(operand1, num_doubles_per_node, MPI_DOUBLE,
              recv_buf1.data(), num_doubles_per_node, MPI_DOUBLE,
              ROOT_RANK, comm);
  MPI_Scatter(operand2, num_doubles_per_node, MPI_DOUBLE,
              recv_buf2.data(), num_doubles_per_node, MPI_DOUBLE,
              ROOT_RANK, comm);

  add(recv_buf1.data(),
      recv_buf2.data(),
      output.data(),
      num_doubles_per_node);

  MPI_Gather(output.data(), num_doubles_per_node, MPI_DOUBLE,
             c, num_doubles_per_node, MPI_DOUBLE,
             ROOT_RANK, comm);
}
```

nur eine Zwischensumme pro Knoten übertragen wird, ist hier allein die Latenz der einschränkende Faktor. Der zweite Nachteil ist, dass das Summieren im Root-Knoten zum Flaschenhals wird, weil dieser Teil nicht mehr verteilt wird. Schließlich kann dort nicht auf Spezialinstruktionen für Skalarprodukte zurückgegriffen werden, etwa *Fused Multiply-Add*, weil die Multiplikation schon abgeschlossen wurde. Die beiden letztgenannten Gründe werden erst bei relativ großen Datenmengen relevant, aber im Sinne der Skalierbarkeit sollte gleich auf diese naive Form der Berechnung verzichtet werden. Dies führt zum hier gezeigten Beispiel. In diesem berechnet jeder Knoten ein lokales Skalarprodukt, was ein optimales Ausnutzen der (Hardware-)Kapazitäten eines jeden Knotens erlaubt. Weiterhin ist es hier analog zu den unabhängigen Operationen möglich, ohne weitere Anpassungen einen autoBLAS-Block für die konkrete Operation einzusetzen. Lediglich das Zusammenführen der Ergebnisse besteht hier nicht aus einem einfachen Senden, denn die Teilergebnisse müssen aufsummiert werden. Der Berechnungsaufwand dieser Summe ist nicht abhängig von der Menge der Daten, sondern von der Anzahl der Knoten in der ausführenden Gruppe, welche üblicherweise weit geringer sein sollte. Einerseits kann hier auf die `MPI_Reduce`-Routine zurückgegriffen werden, die das Senden und gleichzeitig die Aufsummierung der Ergebnisse übernimmt. Eine Alternative besteht darin, die Ergebnisse per Gather auf den Root-Knoten zu übertragen und dort aufzusummieren. Die erste Version ermöglicht eine Optimierung der Kommunikation durch die MPI-Implementierungen. Die zweite Version könnte die Summenoperation wieder als autoBLAS-Block modellieren und somit von einer performanten Implementierung und klareren Darstellung profitieren. Welche der Lösungen performanter ist, hängt von einer Vielzahl an Faktoren ab und müsste gegebenenfalls empirisch in der jeweiligen Umgebung ermittelt werden. Für dieses Beispiel wurde die erste Variante umgesetzt, denn momentan unterstützt autoBLAS keine Summenreduktion.

Der nächste Baustein, der für komplexe Operationen in der linearen Algebra notwendig ist, ist das dyadische Produkt zweier Vektoren. Diese müssen dabei nicht die gleiche Dimension haben, was eine etwas aufwändigere Verwaltungslogik bei der Verteilung mit sich bringen würde. Allerdings hat das dyadische Produkt die Eigenschaft, dass es einen großen Speicherverbrauch hat und dadurch oft nicht explizit berechnet wird. Wenn die Eingabevektoren groß sind, ist es oft sogar nicht einmal *möglich*, es explizit zu bestimmen und zu speichern. In Hinblick auf die Verteilung profitiert das dyadische Produkt kaum von einer Berechnung durch mehrere Knoten, da hauptsächlich Daten kopiert werden müssen. In vielen Anwendungsgebieten ist es allerdings gar nicht notwendig, das dyadische Produkt auszurechnen. Einerseits trifft dies zu, wenn weitere Multiplikationen angehängt werden, da etwa $(\boldsymbol{x} \otimes \boldsymbol{y}) \cdot \boldsymbol{z}$ bei passenden Dimensionen der Vektoren $\boldsymbol{x}$, $\boldsymbol{y}$ und $\boldsymbol{z}$ äquivalent ist zu $\boldsymbol{x}(\boldsymbol{y}^T\boldsymbol{z})$, was sich deutlich speichereffizienter berechnen lässt. Andererseits ist es beispielsweise in der Bildverarbeitung sogar wünschenswert, wenn sich Filtermasken in eine Darstellung als dyadisches Produkt separieren lassen. Aus diesen Gründen wird die Verteilung des dyadischen Produkts in dieser Arbeit nicht weiter betrachtet.

Der folgende Absatz behandelt Matrix-Vektor-Produkte. Für deren Verteilung ist es zunächst wichtig, ob es der Vektor von links oder von rechts mit der Matrix multipliziert wird. Ein Ausdruck wie $\boldsymbol{v}^T\boldsymbol{M}$ kann auf mehrere verschiedene Wege verteilt werden, die folgend näher diskutiert werden, da es keinen Weg gibt, der klar

Listing 4.4: Definition eines Skalarprodukts in autoBLAS

```
    Vector a;
    Vector b;
    Skalar s;
    s = a' * b;
```

Listing 4.5: mögliche Ausgabe von autoBLAS

```
double distributed_dot(const double* a,
                       const double* b,
                       const int length,
                       MPI_Comm comm) {
  constexpr int ROOT_RANK = 0;
  int group_size;
  MPI_Comm_size(comm, &group_size);
  int rank;
  MPI_Comm_rank(comm, &rank);
  const int num_doubles_per_node = length / group_size;

  const double* operand1 = rank == ROOT_RANK ? a : nullptr;
  const double* operand2 = rank == ROOT_RANK ? b : nullptr;
  std::vector<double> output(num_doubles_per_node);

  std::vector<double> recv_buf1(num_doubles_per_node);
  std::vector<double> recv_buf2(num_doubles_per_node);
  MPI_Scatter(operand1, num_doubles_per_node, MPI_DOUBLE,
              recv_buf1.data(), num_doubles_per_node, MPI_DOUBLE,
              ROOT_RANK, comm);
  MPI_Scatter(operand2, num_doubles_per_node, MPI_DOUBLE,
              recv_buf2.data(), num_doubles_per_node, MPI_DOUBLE,
              ROOT_RANK, comm);

  const auto partial_sum =
      dot(recv_buf1.data(), recv_buf2.data(), num_doubles_per_node);

  double total_sum = 0;
  MPI_Reduce(&partial_sum, &total_sum, 1, MPI_DOUBLE,
             MPI_SUM, ROOT_RANK, comm);

  return total_sum;
}
```

besser als die anderen ist. Dafür werden sie zunächst konzeptuell vorgestellt, bevor sie mittels MPI-Routinen skizziert werden.

Das erste Verfahren besteht darin, den Ergebnisvektor elementweise zu berechnen. Dazu muss jedem Knoten eine Kopie des Eingabevektors sowie eine Spalte der Matrix vorliegen. Aus beiden wird auf jedem Knoten ein Skalarprodukt gebildet, das ein Element des Ausgabevektors darstellt. Im letzten Schritt müssen diese Elemente zusammengefügt werden. Für diese erste Variante muss die Anzahl der Spalten der Matrix durch die Anzahl der teilnehmenden Knoten teilbar sein. Die Implementierung dieser Variante kann mittels eines Broadcasts des Eingabevektors nebst eines Scatters der Spalten der Matrix erfolgen. Da Matrizen in C/C++ üblicherweise in der Row-major-Ordnung gespeichert sind und die Scatter-Funktion per se nur das Senden zusammenhängender Speicherbereiche erlaubt, ist hierfür die Definition eines eigenen Vektor-Datentyps notwendig [79, S. 123]. Das Kopieren des Eingabevektors und die komplexere Logik zum Verteilen der Matrix sind Nachteile dieses Ansatzes. Andererseits ist das Berechnen der Einzelergebnisse und das Zusammenfügen mittels `MPI_Gather` sehr einfach.

Das zweite Verfahren basiert auf der zeilenweisen Verteilung der Eingabematrix; hierbei muss also die Zeilenanzahl durch die Knotenzahl teilbar sein. In diesem Fall muss der Eingabevektor nicht auf alle teilnehmenden Knoten kopiert werden, sondern wird derart auf sie aufgeteilt, dass jeder Knoten die zu seinen Zeilen korrespondierenden Elemente erhält. Jeder Knoten berechnet nun wiederum das Matrix-Vektor-Produkt seines Teilvektors mit seiner Teilmatrix, was im einfachsten Fall auf eine Skalarmultiplikation eines Zeilenvektors hinausläuft. Das Ergebnis auf jedem Knoten ist bei dieser Variante eine vektorwertige Teilsumme und muss mit den Ergebnissen der anderen Knoten addiert werden. Wieder davon ausgehend, dass die Matrix in Row-major-Ordnung gespeichert ist, lassen sich ihre Zeilen per Scatter auf die Knoten verteilen. Dabei muss für die Korrektheit sichergestellt werden, dass jeder Knoten eine oder mehrere ganze Zeilen erhält. Die Vektorelemente können ebenfalls per Scatter verteilt werden. Auf jedem Knoten kann nun wieder ein autoBLAS-Block eingefügt werden, der die Teilvektoren mit den Teilmatrizen multipliziert, um einen Teilergebnisvektor zu erhalten. Diese können schließlich per Reduce zusammengeführt werden. In diesem Fall ist die Kommunikationslogik zum Verteilen einfacher als bei der ersten Variante.

Diese beiden Verfahren skalieren nicht über die Anzahl der Spalten beziehungsweise Zeilen der Matrix hinaus. Daher besteht eine letzte Variante in der Kombination beider Verfahren, die die Eingabematrix in Blöcke zerlegt, um die Anzahl der Knoten zu erhöhen, die sich an der Berechnung beteiligen können. Sofern die Blöcke mehrere Zeilen umfassen sollen, sind hierfür wieder benutzerdefinierte Vektordatentypen notwendig. Der Eingabevektor wird in mehrere Teilvektoren gespalten, die wiederum auf die Knoten entsprechend der Zeilenverteilung kopiert werden. Hier berechnet jeder Knoten einige Dimensionen einer vektorwertigen Teilsumme des Endergebnisses. Zur effizienten Ansteuerung der Knoten, die einen bestimmten Teil der Daten erhalten sollen, bietet sich die Nutzung von MPI-Gruppen zur Organisation der Knoten an. Diese Variante ist daher am komplexesten, skaliert jedoch auch am besten mit der Knotenanzahl.

Ein weiterer wichtiger Faktor, der für eine Umsetzung mit autoBLAS berücksichtigt werden muss, ist die Weiterverwendbarkeit von Ergebnissen auf den Knoten. Hierbei geht es darum, inwiefern verhindert werden kann, dass die Ergebnisse zwischen zwei Operationen auf einem Knoten gesammelt und anschließend neu verteilt werden müssen, um den Kommunikationsaufwand zu reduzieren. Ist diese Matrix-Vektor-Multiplikation die finale Operation zum Erhalt eines Ergebnisses, ist dieser Aspekt unwichtig. Bei reduzierenden Operationen wie einer Summenbildung sind alle Ansätze gleichwertig, denn sie können lokale Ergebnisse bestimmen und anschließend kombinieren. Muss das Ergebnis der Matrix-Vektor-Multiplikation mit einem anderen Vektor addiert werden, ist auch dies verteilt ohne Zwischenkommunikation möglich, indem pro Vektordimension nur ein Knoten die Addition vornimmt. Wenn der Ergebnisvektor mit einem Skalar oder elementweise mit einem anderen Vektor multipliziert wird, sind Dank des Distributivgesetzes alle Ansätze verteilt berechenbar, allerdings führen die Ansätze 2 und 3 dann überflüssige Berechnungen durch.

4.4 Legion

Eine mögliche Umsetzung der Verteilung mittels Legion unterscheidet sich deutlich von derjenigen mit MPI. Der Grund dafür ist das taskbasierte Programmiermodell, bei dem die eigentliche Verteilung implizit durch die Legion-Laufzeitumgebung abgewickelt wird. Konkret ist das Programmieren mit Legion um das Erstellen von Tasks zentriert, die bei der Laufzeitumgebung registriert werden müssen und nach der Startanweisung von ihr ausgeführt werden. Jeder Task muss bei der Registrierung eine eindeutige ID zugewiesen werden. Eine Möglichkeit zur Sicherstellung ist die Definition einer `enum` in Legion-Programmen. Alternativ ist es auf verschiedenen Wegen möglich, Task-IDs generieren zu lassen. Beispielsweise gibt es etwa die Methoden `generate_dynamic_task_id` und `generate_library_task_ids` der `Runtime`-Klasse für die dynamische Generierung von IDs oder die statische Methode `generate_static_task_id`, mit der Task-IDs vor dem Start der Laufzeitumgebung generiert werden können.

Start der Laufzeitumgebung Der kritische Punkt bei der Benutzung von Legion ist, dass im Normalfall alle Tasks vor dem Start der Laufzeitumgebung registriert werden müssen und die Laufzeitumgebung dann einmalig initialisiert und gestartet werden muss. Damit ergibt sich ein ähnliches Problem wie bei MPI: Ein solches Vorgehen widerspricht autoBLAS' Modell von lokal zu ersetzenden Blöcken ohne einen globalen Status. Für eine effiziente Implementierung müsste autoBLAS bekannt sein, an welcher Stelle die Laufzeitumgebung einmalig gestartet werden kann. Mit der Einführung eines `#autoblas-main`-Kommandos wäre es zwar möglich, diese Fähigkeit nachzurüsten, doch damit könnten die Benutzbarkeit und Einfachheit vermindert werden. Bisher schafft es autoBLAS im Wesentlichen das Gefühl zu vermitteln, mit geringem Aufwand eine gute Performance zu erreichen. Lediglich die Pflicht zur Nutzung der `#autoblas-include`-Anweisung trübt dieses Bild. Eine `#autoblas-main`-Anweisung wäre im Vergleich damit noch einmal viel schwieriger zu nutzen. Es ist nämlich nicht trivial, an welche Stelle diese Anweisung gesetzt werden muss. Gegen eine Verwendung in der `main`-Funktion spricht, dass autoBLAS mit dieser Anweisung auch eine sogenante Top-Level-Task registrieren muss, die die berechnenden Tasks aufruft. An dieser Stelle ist es jedoch für autoBLAS in den meisten Fällen nicht klar, welche berechnenden Tasks überhaupt aufgerufen werden sollen.

Eine Alternative für die Lösung dieses Problems liegt in der Spezifikation eines Legion-Frontends: Wird autoBLAS mitgeteilt, dass der Code Teil eines Legion-Programms ist, kann es davon ausgehen, dass der Start des Laufzeitsystems manuell erfolgt, und konzeptuell die autoBLAS-Blöcke unabhängig voneinander als Tasks registrieren. In diesem Fall muss der Programmierer sicherstellen, dass Legion erst gestartet wird, nachdem alle Tasks registriert wurden. Auch bei diesem Ansatz gibt es jedoch mehrere Probleme. Einerseits sollte autoBLAS transparent sein und dabei eine gute Performance liefern, das heißt insbesondere vom Programmierer keine Kenntnisse über die konkrete Verteilungslogik erfordern. Andererseits gibt es auch technische Gründe für Schwierigkeiten mit diesem Ansatz: Bei der Registrierung einer Task muss ein Funktionszeiger angegeben werden, mit dem der letztendlich die Task bearbeitende Knoten auf den auszuführenden Code zugreifen kann. Konkreter muss die auszuführende Funktion per Templateparameter spezifiziert werden, was verhindert, dass etwa lokal definierte Lambdaausdrücke als Tasks genutzt werden können. autoBLAS-Blöcke befinden sich allerdings zumeist innerhalb von Funktionen, wodurch es für autoBLAS nicht möglich ist, Objekte (wie Funktionen) außerhalb vom lokalen Scope des autoBLAS-Blockes zu definieren. Dies verhindert wiederum, dass beliebige Knoten des Clusters Zugriff auf die Funktion haben. Schließlich könnte eine autoBLAS-Headerdatei für die Definition der Legion-Tasks verwendet werden. Damit einhergehende Nachteile wurden bereits zuvor besprochen.

4.5 Auswertung

In diesem Kapitel wurden mehrere Frameworks auf ihre technische Passung zur Integration in autoBLAS untersucht. Dabei wurde insbesondere Wert darauf gelegt, wie diejenigen Operationen umgesetzt werden können, die häufig in der linearen Algebra eingesetzt werden. Schließlich findet eine weitere Filterung statt. Vom Funktionsumfang sind sich die untersuchten Frameworks sehr ähnlich. Letztendlich entscheidend für die Auswahl für die Experimente war daher, wie das jeweilige Framework zu initialisieren beziehungsweise zu instrumentieren ist. Eine einmalige, globale Initialisierung ist kaum mit autoBLAS' lokaler Sicht auf unabhängige Blöcke vereinbar. Aus diesem Grund kann Legion für die Experimente nicht verwendet werden. Stattdessen verbleiben MPI und Dask für den folgenden Teil der Arbeit. Letztere Bibliothek musste in diesem Kapitel nicht extra betrachtet werden, da die Verteilung hier implizit umgesetzt wird, während die Schnittstelle aus Nutzersicht (nahezu) identisch zu NumPys ist.

5 Experimente

Typischerweise wird in Papern zuerst die Implementierung beschrieben und die Validität der Lösung anschließend durch Experimente und Messungen verifiziert. In dieser Arbeit finden die Experimente vor der eigentlichen Implementierung statt. In den Experimenten wird daher insbesondere kein von autoBLAS generierter, sondern handgeschriebener Code getestet. Das hängt damit zusammen, dass die Experimente hier einem leicht anderen Zweck dienen. Durch sie soll nicht die hier implementierte Lösung motiviert werden. Stattdessen soll die Implementierung durch die Experimente geleitet werden. Da autoBLAS ein Transpiler ist, lässt sich dieses Vorgehen gut umsetzen, indem die von autoBLAS gewünschte Verteilungslogik für ausgewählte Beispielprogramme per Hand geschrieben und getestet wird. Der BLAS-verwandte Teil, der auf jedem Knoten auf den Teildaten ausgeführt wird, wird allerdings in diesen Experimenten bereits durch das bestehende autoBLAS generiert. Der per Hand geschriebene Code dient insofern auch als Spezifikation für das geplante Verteilungsfeature in dem Sinne, dass der von autoBLAS ausgegebene Code für die Testbeispiele hinterher identisch mit dem händischen sein sollte. Insgesamt hat dieses Vorgehen den Vorteil, Entwicklungszeit für das Einbauen von Verteilungsbackends in autoBLAS einzusparen, die keine akzeptable Leistung erbringen werden. Auf diese Weise lassen sich auch verschiedene Prototypen miteinander vergleichen, um final Entscheidungen für die Implementierung ableiten zu können. Es ist also nicht nötig zu schätzen, welche Implementierung wahrscheinlich am besten abschneiden wird, weil die Daten darüber bereits vorliegen.

In diesem Kapitel werden die verbliebenen Frameworks hinsichtlich ihrer Ausführungsgeschwindigkeit und Skalierbarkeit untersucht. Es ist wie folgt gegliedert. Zunächst wird im Rahmen des Versuchsaufbaus das Szenario beschrieben, in denen die verteilten Berechnungen getestet werden. Anschließend wird auf die Umgebung eingegangen, die verwendet wird, um die Experimente laufen zu lassen. Darin mit eingeschlossen sind eine Beschreibung der verwendeten Software sowie der Hardware, auf der die Programme laufen. Dies dient der Sicherung der Nachvollziehbarkeit der Ergebnisse. Schließlich werden die Ergebnisse beschrieben und diskutiert. Jeglicher Code für die Experimente kann unter `https://git.rz.uni-jena.de/ru76how/experiments` eingesehen werden.

5.1 Versuchsaufbau

5.1.1 Experimentalumgebung

Hardware

Mit Blick auf die Hardware sollten die Experimente für die größtmögliche Aussagekraft auf einem Cluster ausgeführt werden. Aus organisatorischen Gründen konnte dies jedoch im Rahmen dieser Arbeit nicht realisiert werden. Stattdessen wurden die Experimentalergebnisse auf einem System mit einer Intel® Pentium® Silver N6000 und 8GB RAM erhoben. Es ist klar, dass diese Ergebnisse nur bedingt Rückschlüsse auf die Performance auf einem Cluster zulassen. Einerseits ist die CPU viel schwächer als es beispielsweise moderne Prozessoren der Reihen Intel® Xeon® oder AMD EPYC™ in Bezug auf die Taktfrequenz, Anzahl der Kerne oder die Verfügbarkeit von Instruktionssätzen wie AVX sind. Weiterhin sind im Testsystem dieser Arbeit keine GPUs verbaut, was die Leistung im Bereich der linearen Algebra einschränkt. Zuletzt ist auch der verfügbare Arbeitsspeicher weit geringer als bei einem HPC-Cluster, was die Datenmenge reduziert, die in den Experimenten verarbeitet werden kann. Nichtsdestotrotz reicht dieses Testsystem aus, um wichtige Probleme des verteilten autoBLAS-Ansatzes offenzulegen. Auch Tendenzen, welche Konfigurationen performanter sind als andere, lassen sich hieraus bereits ablesen. Der Code für die Experimente selbst ist so geschrieben, dass er auch mit gängigen Mitteln und Programmen wie Slurm auf Clustern ausgeführt werden kann.

Container

In dieser Arbeit werden Container für die Sicherstellung der Reproduzierbarkeit genutzt. Dies ist möglich, indem die für die Experimente notwendige Software zusammen mit ihren Abhängigkeiten verpackt wird und unabhängig vom Hostbetriebssystem und den darauf installierten Softwarebibliotheken und deren Versionen ausgeführt werden kann. In dieser Arbeit wird die Ausführung von der Containerplattform Singularity übernommen, die sich auf die Anforderungen im High-Performance-Computing-Bereich spezialisiert hat [49]. Als solche werden einerseits erweiterte Sicherheitsmechanismen wie kryptografische Signaturen zur Containerverifikation und ein durchdachtes Berechtigungskonzept eingesetzt, das es beispielsweise nicht erlaubt, in Containern weitere Rechte als außerhalb zu erlangen. Weiterhin liegt ein Augenmerk auf der Unterstützung typischer Hardware im HPC wie Hochgeschwindigkeitsnetzwerke, GPUs und parallele Dateisysteme.

Der typische Arbeitsablauf mit Singularity besteht aus zwei Teilen: Zuerst wird ein sogenanntes Image erzeugt, das in einem zweiten Schritt ausgeführt wird. Singularity-Images werden auf Basis einer Definitionsdatei erstellt, die typischerweise die Endung `.def` besitzt. Eine solche ist im Anhang in Listing A.2 abgebildet. Diese spezifiziert etwa, welche Dateien in das Image kopiert werden sollen und welche Kommandos ausgeführt werden müssen, um den gewünschten Status innerhalb des Images herzustellen. Die Erstellung eines Images erfolgt dann mit dem Kommando `singularity build <Image> <Definition>`, das mit erhöhten Rechten ausgeführt werden muss. Die Imagedateien sind grundsätzlich Binärdateien, beginnen jedoch mit einer Shebang-Zeile und werden daher ähnlich wie eine ausführbare Datei behandelt. Wenn eine Imagedatei direkt von der Kommandozeile gestartet wird, wird das Skript innerhalb des `%runscript`-Abschnittes innerhalb der Definitionsdatei ausgeführt. Die flexiblere Möglichkeit zur Ausführung bietet das Kommando `singularity exec <Image> <Cmd> <Args>`, das ein beliebiges Programm starten kann, das im Image vorhanden ist. Somit ist es möglich, mittels `mpirun` ein Programm im Image korrekt auszuführen. `mpirun -n 4 ./image.sif` führt nicht zum richtigen Ergebnis, da hier 4 voneinander getrennte MPI-Instanzen gestartet werden, die nicht miteinander kooperieren können.

Dateistruktur

Die Verzeichnisse für die Experimente dieser Arbeit sind wie folgt aufgebaut. Auf der obersten Ebene liegen jeweils eine Quellcodedatei mit der Endung `.ab`, die autoBLAS-Blöcke enthält, eine `README.md` sowie eine Singularity-Definitionsdatei. In den Ordnern für die Experimente mit C++ finden sich außerdem jeweils eine `CMakeLists.txt`-Datei und ein Makefile für das Bauen. Details zum Bauen und Ausführen eines jeden Experimentes befinden sich in der jeweiligen `README`-Datei.

5.1.2 Problemvorstellung

Die lineare Regression ist ein beliebtes Verfahren, um den Zusammenhang zwischen zwei Eingabegrößen $\boldsymbol{X}$ und $\boldsymbol{y}$ linear zu approximieren. Dabei sind die Zeilen der Matrix $\boldsymbol{X}$ die Beobachtungen und die Einträge im Vektor $\boldsymbol{y}$ die dazugehörigen Messwerte. Gesucht sind die Parameter $\hat{\boldsymbol{\theta}}$ einer optimalen Ausgleichsgeraden, das heißt derjenigen mit dem geringsten quadratischen Fehler (Formel 5.1). Die lineare Regression ist ein einfaches Optimierungsproblem in dem Sinne, dass ein Nullsetzen des Gradienten (Formel 5.2) zu einer geschlossenen Lösung führt (Formel 5.3), falls die Matrix $\boldsymbol{X}^T\boldsymbol{X}$ invertierbar ist. Dies ist der Fall, wenn die Matrix $\boldsymbol{X}$ vollen Rang hat, und lässt sich darüber hinaus durch die Modifikation zu $\boldsymbol{X}^T\boldsymbol{X} \to \boldsymbol{X}^T\boldsymbol{X} + \gamma\boldsymbol{I}$ forcieren.

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}} \|\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}\|^2 \tag{5.1}$$

$$0 \overset{!}{=} \nabla_{\boldsymbol{\theta}} f(\hat{\boldsymbol{\theta}}) = \boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{\theta}} - \boldsymbol{X}^T\boldsymbol{y} \tag{5.2}$$

$$\Leftrightarrow \hat{\boldsymbol{\theta}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{5.3}$$

Das Problem an dieser geschlossenen Formel ist, dass autoBLAS ähnlich wie BLAS selbst Matrixinvertierungen nicht unterstützt [30]. Stattdessen wird für die Experimente ein Gradientenabstiegsverfahren basierend auf dem Gradienten aus Formel (5.2) implementiert. Die resultierende Iterationsvorschrift enthält nach einer vereinfachenden Anwendung des Distributivgesetzes noch Elemente der unteren beiden BLAS-Level (Formel 5.4). Das Level 1 ist durch die Skalarmultiplikation und die beiden Vektorsubtraktionen vertreten, während die beiden Matrix-Vektor-Multiplikationen aus dem Level 2 stammen. Zusätzlich muss als Abbruchbedingung die Formel (5.1) implementiert werden. Das Quadrat der euklidischen Norm darin kann durch das Skalarprodukt abgebildet werden.

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \alpha \boldsymbol{X}^T \big(\boldsymbol{X} \boldsymbol{\theta}_i - \boldsymbol{y} \big) \tag{5.4}$$

Wie eingangs erwähnt wurde, ist das maschinelle Lernen eines der wichtigsten Einsatzfelder von linearer Algebra. Speziell des Deep Learning spielt heutzutage eine zentrale Rolle darin. Für die dort vertretenen Optimierungsaufgaben gibt es üblicherweise keine geschlossenen Formeln, sodass ein Gradientenabstiegsverfahren[1] die Option der Wahl darstellt. Aus diesem Grund wurde auch für die Experimente dieser Arbeit ein Gradientenabstieg ausgewählt. Ein weiteres Argument für die Auswahl dieses Algorithmus' ist, dass iterative Verfahren eine potenzielle Schwäche von autoBLAS darstellen, deren Worst-Case-Performance genauer untersucht werden sollte. Der durch die Iterationen kumulierte Aufwand für das Initialisieren der Verteilungsmechanismen könnte den Vorteil der Verteilung in Abhängigkeit von der Iterationszahl ausgleichen.

Implementierung in autoBLAS Als Ausgangspunkt für die Experimente wurde der Gradientenabstieg für die lineare Regression einerseits mittels C++ und und andererseits mit Python nebst autoBLAS implementiert. Als Beispiel ist in Listing A.1 auf Seite 60 im Anhang die Variante für C++ abgebildet. Das Programm besteht neben `main` aus zwei Funktionen, die die Formeln (5.1) und (5.4) abbilden, sowie der Logik zum Messen der Zeit. Diese Implementierung ist absichtlich nicht optimal gewählt, um einen Nutzer zu simulieren, der wenig bis keine Programmiererfahrung hat und deshalb diese Implementierung wählt, die nah an der mathematischen Formulierung ist. Wesentlich performanter wäre es, beide Formeln innerhalb eines autoBLAS-Blocks umzusetzen, da sie gemeinsame Teilausdrücke besitzen, die dann nur einmal berechnet werden müssten. Als einzige Optimierung wurde das manuelle Einfügen einer temporären Variable vorgenommen, da autoBLAS diese Funktion prinzipiell selbst mitbringt, sie aber zu Abstürzen führt. In beiden Formelfunktionen ist jeweils ein autoBLAS-Block definiert, der im Rahmen der Experimente sowohl durch eine Verteilungslogik als auch einen BLAS-Anteil ersetzt wird. In einer weiteren Funktion wird der Gradientenabstieg realisiert, wobei die Parameter so gewählt wurden, dass das Verfahren konvergiert. Durch die Wahl der Parameter ist außerdem sichergestellt, dass in allen Experimenten dieselbe Anzahl an Iterationen durchlaufen wird. In der `main`-Funktion wird die Zeitmessung mittels einer monotonen Uhr realisiert.

5.2 Auswertung

In diesem Abschnitt werden die Ergebnisse der Experimente vorgestellt. Im Fokus steht insbesondere die Skalierbarkeit bezogen auf die gesamte Ausführungszeit. Als Vergleichswert wird in jedem Fall die benötigte Zeit für die Berechnung mit nur einem Prozess hinzugezogen. Dies dient in Anlehnung an McSherry et al. dazu herauszufinden, wie viel zusätzlicher Aufwand durch die Verteilungslogik entsteht [58].

---

[1] wenn auch nicht in der hier verwendeten, naiven Form; siehe etwa [47]

Tabelle 5.1: Laufzeiten der Experimente in ms

Experiment	Prozessanzahl	Minimalwert	Mittelwert	Maximalwert
`eigen2functions`	1	619	629	645
`numpy2functions`	1	2575	3244	4092
`dask2functions`	1	3704	4305	4687
	2	3645	4103	4529
	4	4159	4452	4617

In jedem Experiment wurden 10 Durchläufe gestartet und deren Zeiten notiert. Die Auflistung der Ergebnisse ist in Tabelle 5.1 abgebildet. Insgesamt lässt sich feststellen, dass die Implementierungen mit C++ denen mit Python bezüglich der Ausführungszeit weit überlegen sind. Im Basisfall (Experiment `eigen2functions`) liegt die mittlere Ausführungszeit bei etwa 629 ms, wobei der längste Durchlauf 645 ms und der kürzeste 619 ms dauerte. Im Vergleich dazu erzielte die Implementierung mittels NumPy in den 10 Durchläufen Werte zwischen 2575 ms und 4092 ms mit dem Mittelwert 3244 ms. Die Implementierung mit C++ ist also nicht lediglich durchschnittlich um den Faktor 5 schneller, sondern auch wesentlich verlässlicher, da die Streuung der Werte viel geringer ist. Ein Großteil der Berechnungen in Python kann zwar auf NumPys effiziente BLAS-Implementierung zurückgreifen, doch durch die interpretierte Ausführung des restlichen Programms kann Python nicht mit C++ mithalten. Auch Dask bringt in der verwendeten Konfiguration keinen Gewinn. Tatsächlich nimmt die in Anspruch genommene Laufzeit im Vergleich mit NumPy sogar zu. Die wahrscheinlichste Ursache dafür ist, dass der verwendete Datensatz zu klein ist, um den Mehraufwand durch Dask zu rechtfertigen. Insbesondere die Schleife spielt als Multiplikator eine große Rolle. Dieser Effekt wird auf Clustern sogar noch verstärkt, da hier in jedem Durchlauf Daten zwischen den Knoten kopiert werden müssen.

Die Experimente mit tatsächlicher MPI-Verteilung konnten bis zuletzt (07. September 2021) nicht durchgeführt werden. Der Grund dafür ist, dass jeder Aufruf von `MPI_Session_init` nach dem ersten einen Speicherzugriffsfehler hervorruft. Es ist unbekannt, ob dies ein Bug in der verwendeten Alphaversion von MPICH 4 ist oder ein beabsichtigtes Verhalten darstellt, das lediglich bisher nicht dokumentiert wurde. Basierend auf den anderen Teilen der Dokumentation wie „In the Sessions Model, MPI resources can be allocated and freed multiple times in an MPI process.“ [79, S. 499] wird hier der erste Fall angenommen. Sobald dieser Fehler in einer Folgeversion behoben wurde, kann auch dieses Experiment evaluiert werden. MPI als Bibliothek für C++ lässt einen deutlichen Geschwindigkeitsvorteil erwarten, bietet dafür allerdings viele Fallstricke, die zu fehlerhaft generierten Programmen führen können. Zudem ist eine komplett statische Verteilung nicht in der Lage, auf dynamische Ungleichverteilungen der Last zu reagieren. Aus diesem Grund sollte autoBLAS sowohl perspektivisch MPI als auch Dask als Verteilungsoptionen anbieten.

6 Integration in autoBLAS

Dieses Kapitel beschreibt, wie die Verteilungsfunktion in autoBLAS implementiert werden kann. Es stellt dazu vor, welche architekturbezogenen Merkmale autoBLAS besitzt, insbesondere vor dem Hintergrund der Compilertheorie. Der zugehörige Code kann unter `https://git.rz.uni-jena.de/ru76how/autoBLAS` abgerufen werden.

6.1 Kurzeinführung in Compilertechnologien

Technisch gesehen ist autoBLAS ein Transpiler, manchmal auch Code-zu-Code-Compiler genannt. Zur besseren Nachvollziehbarkeit der folgenden Abschnitte über die Implementierung der Verteilungsfunktion in autoBLAS gibt dieser Abschnitt einen kurzen Überblick über Compilertechnologien. Ebenfalls aus diesem Grund werden für jeden erläuterten Bestandteil Beispiele aus autoBLAS verwendet.

Compiler bestehen grundsätzlich aus drei Teilen [89, S. 146]. Eine von zwei zentralen Datenstrukturen zur Darstellung von Programmen innerhalb eines Compilers ist der abstrakte Syntaxbaum (AST). Daneben werden Programme häufig in Form eines Kontrollflussgraphen dargestellt, allerdings nicht in autoBLAS. Hier wird ausschließlich ein abstrakter Syntaxbaum verwendet. Er bildet alle sprachlichen Konstrukte in einer umfassenden Datenstruktur ab und enthält üblicherweise verschiedene Statements wie Schleifen, Fallunterscheidungen und Zuweisungen [89, S. 146]. In autoBLAS gibt es neben den Deklarationen, die kein Teil des AST sind, lediglich Zuweisungen, die wiederum hauptsächlich aus Ausdrücken bestehen. Deklarationen werden durch Symboltabellen abgebildet, die Informationen über die Namen und Typen aller deklarierten Variablen enthalten.

Im Frontend des Compilers wird der eingegebene Programmcode in Token zerlegt und der abstrakte Syntaxbaum aufgebaut. Dafür wird in diesem Fall alles außerhalb von `#autoblas`-Anweisungen beziehungsweise -blöcken ignoriert. In autoBLAS wird die Frontendarbeit durch die Bibliothek `Boost.Spirit`[1] übernommen. Damit muss lediglich die gewünschte formale Sprache deklariert werden, während alle weiteren Schritte übernommen werden. Eventuell auftretende Syntaxfehler fallen in diesem Schritt auf und werden dem Benutzer gemeldet. Da die geplanten Änderungen an autoBLAS nur die semantische Ebene betreffen, muss das Frontend nicht modifiziert werden.

Im Kern eines Compilers werden einer oder mehrere Analyse- und Optimierungsschritte auf den AST angewendet [66]. Dies dient dazu, Informationen über das Programm zusammeln, Fehler an den Benutzer zu melden oder die Programmausführung zu beschleunigen. Eine der wichtigsten Aufgaben ist die Durchsetzung des

---

[1] `https://www.boost.org/doc/libs/1_77_0/libs/spirit/doc/html/index.html`

Typsystems. Dieses umfasst in autoBLAS vier Typen: Skalare, Zeilen- und Spaltenvektoren sowie Matrizen. Die Typenprüfung findet nur auf dieser groben Ebene statt, was bedeutet, dass zum Beispiel eine Matrix der Dimension $1 \times n$ nicht als ein Zeilenvektor behandelt wird. autoBLAS bietet einige Optimierungen beziehungsweise Normalisierungen, wobei die meisten nur aktiviert werden, wenn das MKL-Backend ausgewählt wurde. Zu den in jedem Fall aktiven Normalisierungen gehört, dass doppelte Transpositionen oder Negationen eliminiert werden. Eine der speziellen Optimierungen des MKL-Backends ist die Extraktion der skalaren Multiplikation aus Skalarprodukten. Das heißt, es wird die Umformung $\boldsymbol{v}^T (\alpha \boldsymbol{x}) = \alpha \left(\boldsymbol{v}^T \boldsymbol{x}\right)$ verwendet, wodurch `size(x)` $- 1$ Multiplikationen gespart werden.

Zuletzt gibt es das Compilerbackend. Dieses hat die Aufgabe, den abstrakten Syntaxbaum wieder in eine Form zu übersetzen, mit der andere Programme arbeiten können. Beispiele dafür sind das ELF-Format für ausführbare Dateien unter Linux, aber auch wieder menschenlesbarer Programmcode wie in autoBLAS' Fall.

Diese Dreiteilung von Compilern bietet große Vorteile in den Bereichen Flexibilität und Erweiterbarkeit. So wirkt sich eine Änderung der Eingabesyntax nicht darauf aus, welche Optimierungen möglich sind oder welche Ausgabe erzeugt werden kann. Teilweise können sogar neue Funktionen im Frontend implementiert werden, ohne die weiteren Schritte zu beeinflussen, wie dies zuvor am Beispiel der Emulation von `sum(x)` durch $\mathbf{1}^T \boldsymbol{x}$ gezeigt wurde. Darüber hinaus können durch den gemeinsam genutzten abstrakten Syntaxbaum n Frontends und m Backends gleichzeitig unterstützt werden, ohne dass dafür $m \cdot n$ Compiler notwendig sind. Zusätzlich können beliebige Optimierer für jegliche Front- und Backends verwendet werden. Ein Nachteil dieser Architektur ist die kritische Rolle des AST. Wenn ihm Elemente hinzugefügt werden, müssen im Worst-Case-Szenario alle Backends angepasst werden. Durch Abstraktionsschichten in der Software könnte zwar verhindert werden, dass solche Anpassungen vorgenommen werden müssen. Dafür kann beispielsweise das Besuchermuster mit einer leeren Standardimplementierung für einen neuen Knoten im AST verwendet werden. Allerdings müssen Änderungen fast immer auf der semantischen Ebene beachtet werden, damit das Programm korrekt funktioniert.

6.2 Vorüberlegungen

Wie in den vorherigen Teilen dieser Arbeit gezeigt wurde, benötigt die Verteilungsfunktion zwei wesentliche Komponenten. Einerseits ist dies die eigentliche Verteilung. Bei dieser wird der AST für jeden autoBLAS-Block so transformiert, dass er die jeweiligen lokalen Berechnungen abbildet. Dafür muss allerdings kein Code erzeugt werden, denn die konkrete Implementierung des AST kann einem der existierenden Backends überlassen werden. Andererseits muss die Verteilungsfunktion aber auch selbst Code für die Kommunikation generieren. Daraus ergeben sich mehrere Implementierungsmöglichkeiten für die neue Funktion. Eine einfache Variante ist es, die Funktionalität konzeptuell als Ganzes im Backend anzusiedeln. Die Modifikation des AST geschieht in diesem Modell privat; dafür hat das Verteilungsmodul die volle Information über die Transformationen und die Codegenerierung zur Verfügung. Eine weitere Idee ist es, die Verteilungsfunktionalität in Form von zwei Modulen zu implementieren. Eines davon nimmt im Transpilerkern die AST-Transformation vor, während das zweite die Kommunikationsaufrufe im Backend generiert. Ein Vorteil dessen ist, dass der AST nach der Transformation allen weiteren Optimierern und Analysierern im Kern auf eine standardisierte Weise zur Verfügung steht, was die Qualität des generierten Codes verbessern kann. Zwar kann auch ein Backendmodul diese aufrufen, allerdings schadet dies dem klaren Softwaredesign und somit beispielsweise der späteren Modifizierbarkeit des Transpilers an einer zentralen Stelle. Ein Nachteil dieser zweigeteilten Implementierung ist, dass weitere Datenstrukturen nötig wären, die das Kommunikationsmodell abbilden, entsprechend derer der Kommunikationscode generiert wird.

6.3 Implementierung als Meta-Backend

Die Idee hinter dieser Art der Implementierung ist die, dass die Verteilungslogik wie die bereits existierenden Backends aufgebaut ist, aber eines von ihnen als Argument für die konkrete Implementierung auf den einzelnen Knoten annimmt. Damit wird eine gewisse Teilung der Verantwortlichkeiten erreicht, wenn auch nicht so strikt wie bei der Implementierung als zwei getrennte Schritte. Verteilerobjekte modifizieren den abstrakten Syntaxbaum der autoBLAS-Blöcke derart, dass sie die lokalen Berechnungen abbilden, wie sie in den Ausführungen in Kapitel 4 vorgestellt wurden. Zusätzlich wird der zugehörige Kommunikationscode ausgegeben. Anschließend wird auf den beziehungsweise die modifizierten AST das angegebene Backend angewendet, um den Code für die tatsächlichen Berechnungen zu erzeugen. Der Code für diese Erweiterung ist im Branch `feature/meta-backend` zu finden.

Äußerlich kann der Benutzer den gewünschten Distributor analog zum Frontend und Backend per Kommandozeilenargument angeben. Intern verfügt er über die gleiche Schnittstelle wie bestehende Backends. Zusätzlich hat die Distributor-Klasse eine Referenz auf ein Backend. In einem ersten Schritt wird jeder Aufruf einer Methode des Distributors an das zugrundeliegende Backend durchgestellt. In der nächsten Implementierungsstufe werden für jeden autoBLAS-Block die für MPI üblichen Variablen deklariert und initialisiert, namentlich der Rang des jeweiligen Knotens sowie die Größe seiner Gruppe. MPI-Sessions sind zwar noch nicht weit verbreitet und auch nicht fehlerfrei nutzbar, werden aber dennoch hier generiert, da sich autoBLAS-Blöcke in Schleifen sonst nicht umsetzen lassen würden.

Für jede Deklaration eines Vektors oder einer Matrix im autoBLAS-Block generiert das Verteilungsbackend einen Buffer für die lokalen Daten. Dieser wird später zum Empfangen beziehungsweise Versenden der Daten benutzt. Der Einfachheit halber wird ein `std::vector<double>` als Buffer verwendet, denn damit wird der belegte Speicher am Ende des autoBLAS-Blockes automatisch wieder freigegeben. Das `data`-Metadatum der deklarierten Variable wird auf diesen Buffer gesetzt. Zusätzlich werden die Metadaten `size`, `rows` und `cols` angepasst, um die lokalen Werte zu reflektieren. Diese modifizierte Version wird an das BLAS-Backend weitergereicht, um beispielsweise spezifische Variablen zu deklarieren, wie es im Falle von Eigen erforderlich ist. Die eigentliche Verteilung der Operanden eines Ausdrucks wird erst bei Zuweisungen vorgenommen. Dadurch wird sichergestellt, dass kein unnötiger Kommunikationsaufwand entsteht, wenn das Ergebnis nicht verwendet wird. Für die Entscheidung, welche Variablen versendet werden müssen, wird der abstrakte Syntaxbaum des Ausdrucks auf der rechten Seite der Zuweisung durchlaufen und notiert, welche Variablen gelesen werden. Anschließend wird jede davon durch `MPI_Scatter` in die zuvor deklarierten lokalen Buffer übertragen. Es folgt die eigentliche Berechnung mit den lokalen Variablen, deren Code durch das zugrundeliegende BLAS-Backend bestimmt wird. Zuletzt müssen die verteilt vorliegenden Ergebnisse wieder auf dem Hauptknoten gesammelt werden. Dazu wird für jede linke Seite einer Zuweisung ein Aufruf zu `MPI_Gather` generiert. Die Verteilung der anderen Arten von Ausdrücken funktioniert analog beziehungsweise wie im Kapitel 4.3 beschrieben. Durch die derzeitigen Probleme der MPI-Sessions ist eine praktische Anwendung von MPI als Verteilungsbackend ausgeschlossen.

Die Implementierung eines Distributors für Dask gestaltet sich vom Konzept her ähnlich, allerdings entfällt die explizite Kommunikation. Stattdessen müssen durch autoBLAS wie bereits erwähnt Typumwandlungen eingefügt werden, die unter anderem dafür sorgen, dass die Daten verteilt und wieder gesammelt werden. Der Aufruf für das Erzeugen eines Dask-Arrays ist `dask.array.from_array(numpy_array)`. Dies ist allerdings nur für Vektoren und Matrizen notwendig, denn Skalare werden automatisch auf den Zielknoten kopiert. Zum Sammeln der Daten muss in jedem Fall die Methode `compute()` verwendet werden, da Dask Ausdrücke träge auswertet und selbst skalare Werte ohne diesen Methodenaufruf als Dask-Graphen vorliegen.

7 Schlussfolgerungen

7.1 Zusammenfassung

Diese Arbeit ist das Ergebnis einer Untersuchung, inwiefern autoBLAS in verteilten Umgebungen einsetzbar ist. Verteilte Umgebungen bringen einige Herausforderungen mit sich, die es bei einzelnen Computern nicht gibt. Aus diesem Grund sind auch spezielle Technologien erforderlich, um resiliente und effiziente Programme für sie zu schreiben. Forschung und Wirtschaft haben zu diesem Zweck seit mehreren Jahrzehnten verschiedene Bibliotheken, Programmiersprachen und Spracherweiterungen hervorgebracht, um den Umgang zu erleichtern, die Entwicklung zu beschleunigen und die Programmierer vor häufigen Fehlerquellen zu bewahren.

Diese Ziele verfolgt auch autoBLAS, setzt allerdings auf einer höheren Ebene an. Das Einlesen und Ersetzen von markierten Blöcken im Quellcode bietet eine Flexibilität, die durch die anderen Technologien nicht erreicht werden kann. Zudem ist die Eingabesprache so leicht zu lernen und zu verstehen, dass autoBLAS einem breiten Anwenderkreis zur Verfügung steht.

Vor dieser Arbeit war es autoBLAS nicht direkt möglich, verteilt funktionierenden Code zu produzieren. Um diese Funktion nachzurüsten, wurden in dieser Arbeit folgende Schritte durchgeführt. Zunächst wurde untersucht, welche Bibliotheken beziehungsweise Frameworks es zum Verteilen von linearer Algebra gibt. Diese wurden hinsichtlich ihrer Fähigkeiten, Ansätze, Passung zu autoBLAS' Modell und Softwarequalitätsmerkmale bewertet, um diejenigen Kandidaten auszuwählen, die sich als Verteilungsbackend für autoBLAS eignen. Anschließend wurde gezeigt, mit welchen Programmkonstrukten Ausdrücke der linearen Algebra in den untersuchten Frameworks repräsentiert werden können. Dabei wurde auch allgemeines Optimierungspotenzial erkundet, etwa, welche Operationen sich mit einem Minimum an Kommunikation umsetzen lassen. Es folgte eine empirische und experimentgetriebene Untersuchung, welche Ansätze zu einer guten Performance des generierten Codes führen. Diese wurden hinzugezogen, um die Implementierung der Verteilungsfunktion in autoBLAS anzuleiten, welche im letzten Kapitel skizziert wurde.

7.2 Ausblick

Ein komplexes Projekt wie dieses bietet zahlreiche Ansatzpunkte für Erweiterungen und Verbesserungen. Einige davon wurden bereits in den vorherigen Kapiteln dieser Arbeit genannt. Dazu gehört beispielsweise die Einführung von autoBLAS-Zusatzdateien, etwa in Form von Headern im Umfeld von C++. Für diese ist noch nicht klar, wie sie in die aktuelle Referenzimplementierung integriert werden können, aber sie bieten das Potenzial, einige der Beschränkungen durch die lokale Sicht zu mindern. Diese könnten unter anderem den Weg zu dynamischen Laufzeitumgebungen ebnen, die autoBLAS etwas Verteilungsaufwand zur Zeit der statischen Programmanalyse abnehmen und in die Laufzeit verschieben, wenn mehr Informationen zur Verfügung stehen. Hierbei ist es beispielsweise nicht notwendig, dass autoBLAS globale Informationen über den Aufrufgraphen zur Verfügung hat, da dieser von den Verteilungssystemen dynamisch aufgebaut werden kann. Basierend auf der theoretischen Analyse sind HPX und Legion besonders erfolgsversprechende Kandidaten dafür, da sie noch immer aktiv entwickelt werden und sich verhältnismäßig leicht aus C++ heraus nutzen lassen.

Bezüglich der Datenerhebung sind detailliertere Einblicke sinnvoll, um fundiertere Entscheidungen über die Implementierung der Verteilungsfunktion treffen zu können. In dieser Arbeit konnte lediglich die Gesamtlaufzeit erhoben werden. Zusätzlich lohnenswerte Performanceindikatoren sind die Auslastung von CPUs und GPUs beziehungsweise deren Wartezeiten. In dieser Arbeit wurde mit der Iteration ein Testszenario gewählt, das relativ schwierig für autoBLAS umzusetzen ist. Darüber hinaus sollten allerdings auch weitere Testszenarien durchgeführt werden. So fehlt beispielsweise ein Szenario mit nur einem autoBLAS-Block, der dafür sehr umfangreich ist. In diesem wird eine bestmögliche Performance durch autoBLAS' Verteilungsfunktionalität erwartet, wodurch es sich für Aussagen zur Maximalleistung eignet.

In dieser Arbeit lag der Fokus darauf, die Daten der Berechnung zu verteilen, die dies insbesondere Vorteile bei großen Datensätzen bietet. Zukünftige Forschung könnte das Augenmerk jedoch auch mehr auf die Verteilung von Instruktionen legen. Hierfür bietet sich die Nutzung eines Kontrollflussgraphen innerhalb von autoBLAS an, um Abhängigkeiten zwischen den verwendeten Variablen aufzudecken.

7.3 Bewertung

Insgesamt zeichnet sich ab, dass eine Verteilung von Ausdrücken der linearen Algebra auf Cluster nicht ideal zur Ausrichtung von autoBLAS in seiner derzeitigen Form passt. Der Grund für diese Diskrepanz ist die vielfach angesprochene Einschränkung von autoBLAS auf lokale und voneinander unabhängige Blöcke. Einerseits verhindert dies die Nutzung von Bibliotheken, die eine einmalige und globale Initialisierung erfordern. Andererseits wirkt sich dies negativ auf die Informationen aus, die autoBLAS für die Entscheidungen zur Verteilung von Daten zur Verfügung hat. Nichtsdestotrotz ist es mittels Dask gelungen, Berechnungen für ein von autoBLAS unterstütztes Backend zu verteilen. Auch für MPI wurde konzeptuell beschrieben, wie die Verteilung im Rahmen von autoBLAS umgesetzt werden kann. FÜr den praktischen Einsatz muss das Session-Modell allerdings erst die Fehler der Alphaphase beheben. Die erzielten Ergebnisse lassen vermuten, dass die Verteilungsfunktion keinen großen Mehrwert bietet, wenn die verwendeten Datenmengen zu gering sind und wenn der verteilte Code in Schleifen ausgeführt wird. Mit zukünftigen Erweiterungen ist es jedoch möglich, dass auch Personen ohne Programmierkenntnisse Zugang zum Hochleistungsrechnen durch autoBLAS erhalten werden.

Literatur

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu und W. Zwaenepoel. „TreadMarks: Shared Memory Computing on Networks of Workstations“. In: *Computer* 29 (März 1996), S. 18–28.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov und D. Sorensen. *LAPACK users' guide: Release 1.0*. `https://www.osti.gov/biblio/10136133`. (letzter Zugriff: 2021-09-07). Jan. 1992.

[3] J. Bachan, D. Bonachea, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, B. van Straalen und S. B. Baden. „The UPC++ PGAS Library for Exascale Computing“. In: *Proceedings of the Second Annual PGAS Applications Workshop*. PAW17. Denver, CO, USA: Association for Computing Machinery, 2017.

[4] B. Barker. „Message passing interface (MPI)“. In: *Workshop: High Performance Computing on Stampede*. Bd. 262. 2015.

[5] M. Bauer. „Legion: Programming Distributed Heterogeneous Architectures with Logical Regions“. Diss. Stanford University, Dez. 2014.

[6] M. Bauer und M. Garland. „Legate NumPy: Accelerated and Distributed Array Computing“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019.

[7] M. Bauer, S. Treichler, E. Slaughter und A. Aiken. „Legion: Expressing Locality and Independence with Logical Regions“. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012.

[8] M. Bauer, S. Treichler, E. Slaughter und A. Aiken. „Structure Slicing: Extending Logical Regions with Fields“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, S. 845–856.

[9] D. M. Beazley. „SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++“. In: *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*. TCLTK'96. Monterey, California: USENIX Association, 1996, S. 15.

[10] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley und Y. Bengio. „Theano: A CPU and GPU Math Compiler in Python“. In: *Proceedings of the 9th Python in Science Conference*. Hrsg. von S. van der Walt und J. Millman. 2010, S. 18–24.

[11] J. Bezanson, A. Edelman, S. Karpinski und V. B. Shah. „Julia: A fresh approach to numerical computing“. In: *SIAM Review* 59.1 (2017), S. 65–98.

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall und Y. Zhou. „Cilk: An Efficient Multithreaded Runtime System“. In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPOPP ’95. Santa Barbara, California, USA: Association for Computing Machinery, 1995, S. 207–216.

[13] R. D. Blumofe und C. E. Leiserson. „Scheduling Multithreaded Computations by Work Stealing“. In: *Journal of the ACM* 46.5 (Sep. 1999), S. 720–748.

[14] D. Bonachea und P. H. Hargrove. „GASNet-EX: A High-Performance, Portable Communication Library for Exascale“. In: *Proceedings of Languages and Compilers for Parallel Computing (LCPC’18).* Bd. 11882. Lecture Notes in Computer Science. Lawrence Berkeley National Laboratory Technical Report (LBNL-2001174). Springer International Publishing, Okt. 2018.

[15] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault und J. J. Dongarra. „PaRSEC: Exploiting Heterogeneity to Enhance Scalability“. In: *Computing in Science Engineering* 15.6 (2013), S. 36–45.

[16] G. Calin, E. Derevenetc, R. Majumdar und R. Meyer. „A Theory of Partitioned Global Address Spaces“. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013).* Hrsg. von A. Seth und N. K. Vishnoi. Bd. 24. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013, S. 127–139.

[17] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks und K. Warren. *Introduction to UPC and language specification. Technical Report CCS-TR-99-157.* Techn. Ber. IDA Center for Computing Sciences, 1999.

[18] T. Carneiro, N. Melab, A. Hayashi und V. Sarkar. „Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators“. In: *HPCS 2020 - The 18th International Conference on High Performance Computing & Simulation.* Barcelona / Virtual, Spain, März 2021.

[19] B. Chamberlain, D. Callahan und H. Zima. „Parallel Programmability and the Chapel Language“. In: *The International Journal of High Performance Computing Applications* 21.3 (2007), S. 291–312.

[20] B. Chamberlain, S.-E. Choi, C. Lewis, C. Lin, L. Snyder und D. Weathersby. „ZPL: A Machine Independent Programming Language for Parallel Computers“. In: *IEEE Transactions on Software Engineering* 26.3 (2000), S. 197–211.

[21] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel und L. Smith. „Introducing OpenSHMEM: SHMEM for the PGAS Community“. In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model.* PGAS ’10. New York, New York, USA: Association for Computing Machinery, 2010.

[22] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun und V. Sarkar. „X10: An Object-Oriented Approach to Non-Uniform Cluster Computing“. In: *SIGPLAN Not.* 40.10 (Okt. 2005), S. 519–538.

[23] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker und R. C. Whaley. „ScaLAPACK: A portable linear algebra library for distributed memory computers — Design issues and performance“. In: *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science.* Hrsg. von J. Dongarra, K. Madsen und J. Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, S. 95–106.

[24] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker und R. C. Whaley. „A proposal for a set of parallel basic linear algebra subprograms“. In: *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science.* Hrsg. von J. Dongarra, K. Madsen und J. Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, S. 107–114.

[25] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[26] *cuBLAS.* `https://docs.nvidia.com/cuda/cublas/index.html`. (letzter Zugriff: 2021-09-07).

[27] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken und K. Yelick. „Parallel Programming in Split-C“. In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing.* Supercomputing '93. Portland, Oregon, USA: Association for Computing Machinery, 1993, S. 262–273.

[28] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem und W. De Meuter. „Partitioned Global Address Space Languages“. In: *ACM Comput. Surv.* 47.4 (Mai 2015).

[29] N. Denoyelle, B. Goglin, E. Jeannot und T. Ropars. „Data and Thread Placement in NUMA Architectures: A Statistical Learning Approach“. In: *Proceedings of the 48th International Conference on Parallel Processing.* ICPP 2019. Kyoto, Japan: Association for Computing Machinery, 2019.

[30] J. J. Dongarra, J. Du Croz, S. Hammarling und I. S. Duff. „A Set of Level 3 Basic Linear Algebra Subprograms“. In: *ACM Trans. Math. Softw.* 16.1 (März 1990), S. 1–17.

[31] J. J. Dongarra, J. Du Croz, S. Hammarling und R. J. Hanson. „An extended set of FORTRAN basic linear algebra subprograms“. English. In: *ACM Transactions on Mathematical Software* 14.1 (1988), S. 1–17.

[32] J. J. Dongarra, P. Luszczek und A. Petitet. „The LINPACK Benchmark: past, present and future“. In: *Concurrency and Computation: Practice and Experience* 15.9 (2003), S. 803–820. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728`.

[33] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally und P. Hanrahan. „Sequoia: Programming the Memory Hierarchy“. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing.* SC ’06. Tampa, Florida: Association for Computing Machinery, 2006, 83–es.

[34] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham und T. S. Woodall. „Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation“. In: *Proceedings, 11th European PVM/MPI Users’ Group Meeting.* Budapest, Hungary, Sep. 2004, S. 97–104.

[35] G. A. Geist, J. A. Kohl und P. M. Papadopoulos. „PVM and MPI: a Comparison of Features“. In: *Calculateurs Paralleles* 8 (1996), S. 137–150.

[36] K. Goto und R. Van De Geijn. „Anatomy of High-Performance Matrix Multiplication“. In: *ACM Trans. Math. Softw.* 34.3 (Mai 2008).

[37] K. Goto und R. Van De Geijn. „High-Performance Implementation of the Level-3 BLAS“. In: *ACM Trans. Math. Softw.* 35.1 (Juli 2008).

[38] D. Grünewald und C. Simmendinger. „The GASPI API specification and its implementation GPI 2.0“. In: *Proceedings of the 7th International Conference on PGAS Programming Models.* Hrsg. von M. Weiland, A. Jackson und N. Johnson. United Kingdom: University of Edinburgh, Okt. 2013, S. 243–248.

[39] G. Guennebaud, B. Jacob et al. *Eigen v3.* `https://eigen.tuxfamily.org`. (letzter Zugriff: 2021-09-07). 2010.

[40] R. Hanson, F. Krogh und C. Lawson. „A proposal for standard linear algebra subprograms“. In: *ACM SIGNUM Newsletter* 8 (1973).

[41] C. R. Harris et al. „Array programming with NumPy“. In: *Nature* 585.7825 (Sep. 2020), S. 357–362.

[42] Helmholtz Zentrum Dresden Rossendorf. *Aufbruch in die Exaflops-Welt.* `https://www.hzdr.de/db/Cms?pOid=59532`. (letzter Zugriff: 2021-09-07). 2019.

[43] C.-C. Huang, Q. Chen, Z. Wang, R. Power, J. Ortiz, J. Li und Z. Xiao. „Spartan: A Distributed Array Framework with Smart Tiling“. In: *2015 USENIX Annual Technical Conference (USENIX ATC 15).* Santa Clara, CA: USENIX Association, Juli 2015, S. 1–15.

[44] Intel Corporation. *Intel® oneAPI Math Kernel Library (oneMKL) – Developer Reference.*

[45] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei und T. Zhang. „HPX - The C++ Standard Library for Parallelism and Concurrency“. In: *Journal of Open Source Software* 5.53 (2020), S. 2352.

[46] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio und D. Fey. „HPX: A Task Based Programming Model in a Global Address Space“. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS ’14. Eugene, OR, USA: Association for Computing Machinery, 2014.

[47] D. P. Kingma und J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: `1412.6980 [cs.LG]`.

[48] M. Kleppmann. *Designing Data-Intensive Applications. The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Hrsg. von A. Spencer und M. Beaugureau. Beijing, Boston, Farnham, Sebastopol, Tokyo: O’Reilly Media, Inc., 2017.

[49] G. Kurtzer, V. Sochat und M. Bauer. *Singularity: Scientific containers for mobility of compute*. PLoS ONE 12(5): e0177459. 2017.

[50] C. Lattner und V. Adve. „LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation“. In: *CGO*. San Jose, CA, USA, März 2004, S. 75–88.

[51] S. Laue, M. Mitterreiter und J. Giesen. „A Simple and Efficient Tensor Calculus“. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34 (Apr. 2020), S. 4527–4534.

[52] S. Laue, M. Mitterreiter und J. Giesen. „Computing Higher Order Derivatives of Matrix and Tensor Expressions“. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS’18. Montréal, Canada: Curran Associates Inc., 2018, S. 2755–2764.

[53] S. Laue, M. Mitterreiter und J. Giesen. „GENO – GENeric Optimization for Classical Machine Learning“. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019.

[54] C. Lawson, R. Hanson, D. R. Kincaid und F. Krogh. „Basic Linear Algebra Subprograms for Fortran Usage“. In: *ACM Trans. Math. Softw.* 5.3 (Sep. 1979), S. 308–323.

[55] W. Lee. „A Hybrid Approach to Automatic Program Parallelization via Efficient Tasking with Composable Data Partitioning“. Diss. Stanford University, Dez. 2019.

[56] R. Martin. *Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code ; [Kommentare, Formatierung, Strukturierung ; Fehler-Handling und Unit-Tests ; zahlreiche Fallstudien, Best Practices, Heuristiken und Code Smells]*. mitp Professional. mitp, 2009.

[57] P. McCormick, C. Sweeney, N. Moss, D. Prichard, S. K. Gutierrez, K. Davis und J. Mohd-Yusof. „Exploring the Construction of a Domain-Aware Toolchain for High-Performance Computing“. In: *Proceedings of the 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. WOLFHPC ’14. USA: IEEE Computer Society, 2014, S. 1–10.

[58] F. McSherry, M. Isard und D. G. Murray. „Scalability! But at what COST?“ In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, Mai 2015.

[59] G. Mencagli, F. M. França, C. B. Bentes, L. A. Justen Marzulo, M. Lima Pilla, R. Wyrzykowski, E. Deelman, C. Simmendinger, R. Iakymchuk, L. Cebamanos, D. Akhmetova, V. Bartsch, T. Rotaru, M. Rahn, E. Laure und S. Markidis. „Interoperability Strategies for GASPI and MPI in Large-Scale Scientific Applications“. In: *Int. J. High Perform. Comput. Appl.* 33.3 (Mai 2019), S. 554–568.

[60] M. Merrill, W. Reus und T. Neumann. „Arkouda: Interactive Data Exploration Backed by Chapel“. In: *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*. CHIUW 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, S. 28.

[61] A. Møller und M. I. Schwartzbach. *Static Program Analysis*. `https://cs.au.dk/~amoeller/spa/`. (letzter Zugriff: 2021-09-07). Juni 2021.

[62] J. Nieplocha, V. Tipparaju, M. Krishnan und D. K. Panda. „High Performance Remote Memory Access Communication: The Armci Approach“. In: *The International Journal of High Performance Computing Applications* 20.2 (2006), S. 233–253.

[63] J. Nieplocha, R. J. Harrison und R. J. Littlefield. „Global Arrays: A Portable 'Shared-Memory' Programming Model for Distributed Memory Computers“. In: *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. Supercomputing '94. Washington, D.C.: IEEE Computer Society Press, 1994, S. 340–349.

[64] R. W. Numrich und J. Reid. „Co-Array Fortran for Parallel Programming“. In: *SIGPLAN Fortran Forum* 17.2 (Aug. 1998), S. 1–31.

[65] L. Page, S. Brin, R. Motwani und T. Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab, Nov. 1999.

[66] D. Quinlan und C. Liao. „The ROSE source-to-source compiler infrastructure“. In: *Cetus users and compiler infrastructure workshop, in conjunction with PACT*. Citeseer. 2011.

[67] E. S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[68] J. Reid, B. Long und J. Steidel. „History of Coarrays and SPMD Parallelism in Fortran“. In: *Proc. ACM Program. Lang.* 4.HOPL (Juni 2020).

[69] C. Rice University. „High Performance Fortran Language Specification“. In: *SIGPLAN Fortran Forum* 12.4 (Dez. 1993), S. 1–86.

[70] A. D. Robison. „Composable Parallel Patterns with Intel Cilk Plus“. In: *Computing in Science and Engineering* 15.2 (März 2013), S. 66–71.

[71] M. Rocklin. „Dask: Parallel Computation with Blocked algorithms and Task Scheduling“. In: *Proceedings of the 14th Python in Science Conference*. Hrsg. von K. Huff und J. Bergstra. 2015, S. 130–136.

[72] A. Sidelnik, S. Maleki, B. Chamberlain, M. Garzarán und D. Padua. „Performance Portability with the Chapel Language". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (2012), S. 582–594.

[73] R. L. Sites. „An Analysis of the Cray-1 Computer". In: *Proceedings of the 5th Annual Symposium on Computer Architecture*. ISCA '78. New York, NY, USA: Association for Computing Machinery, 1978, S. 101–106.

[74] E. Slaughter. „Regent: A High-Productivity Programming Language for Implicit Parallelism with Logical Regions". Diss. Stanford University, Aug. 2017.

[75] E. Slaughter, W. Lee, S. Treichler, M. Bauer und A. Aiken. „Regent: A High-Productivity Programming Language for HPC with Logical Regions". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: Association for Computing Machinery, 2015.

[76] E. Slaughter, W. Lee, S. Treichler, W. Zhang, M. Bauer, G. Shipman, P. McCormick und A. Aiken. „Control Replication: Compiling Implicit Parallelism to Efficient SPMD with Logical Regions". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: Association for Computing Machinery, 2017.

[77] V. S. Sunderam. „PVM: A Framework for Parallel Distributed Computing". In: *Concurrency: Pract. Exper.* 2.4 (Nov. 1990), S. 315–339.

[78] The MPI Forum. „MPI: A Message Passing Interface". In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA: Association for Computing Machinery, 1993, S. 878–883.

[79] The MPI Forum. *MPI: A Message-Passing Interface Standard. Version 4.0.* `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`. (letzter Zugriff: 2021-09-07). 2021.

[80] Thinking Machines Corporation. *CM-5 C* User Guide. Version 7.1.* `http://people.csail.mit.edu/bradley/cm5docs/CM-5CStarUsersGuide.pdf`. (letzter Zugriff: 2021-09-07). Cambridge, Massachusetts, Mai 1993.

[81] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure und D. S. Nikolopoulos. „A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing". In: *J. Supercomput.* 74.4 (Apr. 2018), S. 1422–1434.

[82] R. Tohid, B. Wagle, S. Shirzad, P. Diehl, A. Serio, A. Kheirkhahan, P. Amini, K. Williams, K. Isaacs, K. Huck und et al. „Asynchronous Execution of Python Code on Task-Based Runtime Systems". In: *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)* (Nov. 2018).

[83] TOP500. *The List.* `https://www.top500.org/lists/top500/list/2021/06/`. (letzter Zugriff: 2021-09-07). Juni 2021.

[84] S. Treichler, M. Bauer und A. Aiken. „Language Support for Dynamic, Hierarchical Data Partitioning“. In: *SIGPLAN Not.* 48.10 (Okt. 2013), S. 495–514.

[85] U.S. Department of Energy. *U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL.* `https://www.energy.gov/articles/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl`. (letzter Zugriff: 2021-09-07). 2019.

[86] M. van Steen und A. Tanenbaum. *Distributed Systems.* 3. Aufl. Version 03. Maarten van Steen, 2020.

[87] M. Véstias und H. Neto. „Trends of CPU, GPU and FPGA for high-performance computing“. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL).* 2014, S. 1–6.

[88] R. C. Whaley und J. J. Dongarra. „Automatically Tuned Linear Algebra Software“. In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing.* SC ’98. San Jose, CA: IEEE Computer Society, 1998, S. 1–27.

[89] N. Wirth. *Grundlagen und Techniken des Compilerbaus.* 3. Aufl. München: Oldenbourg Verlag, 2011.

[90] Z. Xianyi, W. Qian und Z. Yunquan. „Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor“. In: *2012 IEEE 18th International Conference on Parallel and Distributed Systems.* 2012, S. 684–691.

[91] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella und A. Aiken. „Titanium: A High-Performance Java Dialect“. In: *ACM 1998 Workshop on Java for High-Performance Network Computing.* New York, NY 10036, USA: ACM Press, 1998.

[92] T. Yuasa, T. Kijima und Y. Konishi. „The data-parallel C language NCX and its implementation strategies“. In: *Theory and Practice of Parallel Programming.* Hrsg. von T. Ito und A. Yonezawa. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, S. 433–456.

Tabellenverzeichnis

Abbildungsverzeichnis

A Anlagen

A.1 Quellcode

Listing A.1: Lineare Regression per Gradientenabstieg in autoBLAS

```
#autoblas-include
#include <chrono>
#include <iostream>
#include <vector>

void gradient_descent_iter(const double* variates,
                           double* params,
                           const double* covariates,
                           const double a,
                           const size_t num_observations,
                           const size_t dims)
{
#autoblas {
  Matrix X data=variates stride=1 rows=num_observations cols=dims ld=1;
  Vector t data=params stride=1 size=dims;
  Vector y data=covariates stride=1 size=num_observations;
  Scalar a;
  t = t - a * X'*(X*t - y);
}
}

double gradient_descent_loss(const double* variates,
                             const double* params,
                             const double* covariates,
                             const size_t num_observations,
                             const size_t dims)
{
  // necessary to do manually; otherwise autoBLAS will crash with MKL backend
  std::vector<double> tmp(num_observations);
  double loss = 0;
#autoblas {
  Matrix X_ data=variates stride=1 rows=num_observations cols=dims ld=1;
  Vector t_ data=params stride=1 size=dims;
  Vector y_ data=covariates stride=1 size=num_observations;
  Vector tmp_ data=tmp.data() stride=1 size=num_observations;
  Scalar loss;
  tmp_ = X_ * t_ - y_;
  loss = tmp_' * tmp_;
}
return loss;
}
```

```
// returns final loss
double linreg_minimize(const double* variates, const double* covariates, double* params, const size_t num_observations, const size_t num_dims)
{
  // arbitrarily chosen
  const double target = 5;
  const double alpha = 0.000000015;

  int iteration_count = 1;
  double loss = gradient_descent_loss(
    variates, params, covariates, num_observations, num_dims);

  while (loss > target) {
    gradient_descent_iter(variates,
                          params,
                          covariates,
                          alpha,
                          num_observations,
                          num_dims);
    loss = gradient_descent_loss(
      variates, params, covariates, num_observations, num_dims);
  }
  return loss;
}

int main(void) {
  const size_t num_observations = 1337;
  const size_t dims = 0xc0de;
  const std::vector<double> variates(num_observations * dims, 1);
  const std::vector<double> covariates(num_observations, 1);
  std::vector<double> params(dims, 2);

  using namespace std::chrono;
  const auto start = steady_clock::now();
  const double loss = linreg_minimize(variates.data(), covariates.data(), params.data(), num_observations, dims);
  const auto end = steady_clock::now();
  const auto duration = duration_cast<milliseconds>(end-start).count();
  std::cout << "computation took " << duration << " ms to find a solution with loss " << loss << std::endl;
  return 0;
}
```

A.2 Singularity-Definitionsdatei

Listing A.2: `linreg.def` für das `eigen2functions`-Experiment

```
Bootstrap: docker
From: ubuntu:latest

%files
  ./CMakeLists.txt /linear_regression/CMakeLists.txt
  ./eigen2functions.cpp /linear_regression/eigen2functions.cpp
```

```

%post
  export DEBIAN_FRONTEND=noninteractive
  apt update -y
  apt install -y g++ cmake libeigen3-dev
  cmake -E make_directory /linear_regression/build
  cmake -DCMAKE_BUILD_TYPE=Release -B /linear_regression/build -S /
     linear_regression
  cmake --build /linear_regression/build

%runscript
  /linear_regression/build/eigen2functions
```

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.
Seitens des Verfassers bestehen keine Einwände die vorliegende Masterarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Kahla, den 8. September 2021 Mark Umnus