TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR RECHNERARCHITEKTUR
PROF. DR. WOLFGANG E. NAGEL

Proseminar “Rechnerarchitektur”

Julia - High-Performance Programmiersprache für High-Performance Computing?

Paul Gottschaldt
(Mat.-Nr.: 4609055)

Hochschullehrer: Prof. Dr. Wolfgang E. Nagel
Betreuer: Martin Schroschk

Dresden, 9. Juli 2018

Inhaltsverzeichnis

1 Einleitung

1998 startete das Apache-Point-Observatorium [Wik18b] in New Mexico (USA) ihr Projekt Sloan Digital Sky Survey (SDSS) [Wik18f] [Ast14], welches Fotos von über 35 % aller sichtbaren Objekte unseres Himmels anfertigte und in einer Datenbank zunächst sicherte. Seit diesem Jahr wurden bereits über 500 Millionen Sterne und Galaxien fotografiert und Licht eingefangen, welches bereits Milliarden von Jahren unterwegs war und die Forscher bis weit in die Vergangenheit unseres Universums zurückblicken lässt. Die SDSS-Kamera galt bis zu ihrer Abschaltung im Jahr 2009 als produktivste Weltraumkamera der Welt. Mit etwa 200 GigaByte reine Bilddaten pro Nacht entstand so eine Datenbank mit über 5 Millionen Bildern von jeweils 12 MegaByte - zusammengerechnet also rund 55 Terabyte [JC16].
2014 startete ein Team von Astronomen, Physikern, Informatikern und Statistikern das Projekt Celeste, um eben jenen Datensatz zu katalogisieren und für jeden Himmelskörper einen Eintrag anzulegen [JC17]. In der ersten veröffentlichten Version von 2015 noch auf Berechnungen auf einzelne Knoten beschränkt, schaffte es das Forschungsteam um die involvierten Parteien Julia Computing, UC Berkeley, Intel, National Energy Research Scientific Computing Center (NERSC) und Lawrence Berkeley National Laboratory in der zweiten Version von 2016 bereits einen 225-fachen Geschwindigkeitsgewinn zu erzielen. Die größten Verbesserungen waren dabei der mit 8192 Intel® Xeon® Prozessoren ausgestattete Hochleistungsrechner (zu engl. High-Performance-Computer oder HPC) des Berkeley Lab's und Julia, eine High-Performance Open-Source Programmiersprache, die bis dato noch relativ unbekannt war und seit 2009 am MIT entwickelt wird. [hei16] [Pra17] Besonders ist an Julia vorallem ihr Anspruch eine sehr produktive Programmiersprache, wie Python zu sein, dabei aber Performanz wie C zu erreichen.
Im November 2017 veröffentlichte dieses Team dann die aktuelle dritte Version, mit der sie nochmals einen deutlichen Geschwindigkeitsgewinn erreichen konnten. Sie schafften es 188 Millionen Sterne und Galaxien in nur 14,6 Minuten zu katalogisieren und nutzten dafür bis zu 1,3 Millionen Threads auf den 9300 Knights Landing Knoten des Cori Supercomputers des NERSC. Dabei erzielten sie mit Julia eine Peak-Performance von 1,54 Petaflop pro Sekunde für Berechnungen mit doppelter Genauigkeit. Damit erzielte das Celeste Projekt einen 100-fachen Geschwindigkeitsgewinn zu allen vorherigen Forschungsprojekten. Derzeit gibt es rund 200 Supercomputer in der Welt, welche in der Lage sind eine Peak-Performance von mehr als einem Petaflop pro Sekunde zu erreichen. Trotzdem sind so gut wie alle Anwendungen, welche eben jene Peak-Performance erreichen von wenigen Gruppen von Experten geschrieben, welche ein sehr tiefes Verständnis für alle erforderlichen Details besitzen [Far17]. Umso erstaunlicher ist an diesem Ergebnis also die Tatsache, dass das Team hinter Celeste die Geschwindigkeit ausschließlich unter Nutzung ihrer Kenntnisse, dem Julia-Code und dessen Threading-Model erzielen konnten.
In Abbildung 1 ist die pro-Thread-Auslastung des Celeste Projekts in Version 3 dargestellt. Zu sehen ist, dass Julia 82,3 % der Zeit lief. Damit stellt Julia unter Beweis, das es sowohl für TheraByte-große Datensätze, als auch für Petaflop pro Sekunde-schnelle Berechnungen geeignet ist.
Keno Fischer, CTO von Julia Computing, sagte 2017, dass Projekte wie Celeste unter Beweis stellen, das der Traum, der hinter Julia steht, wahr wird. Wissenschaftler können nun auf ihrem Rechner entwickelte Prototypen einfach von ihrem Laptop auf die größten Supercomputer verschieben ohne zwischen Sprachen wechseln zu müssen oder ihren Code gar komplett neu zu schreiben. Sie (Anmerkung: Entwickler von Julia Computing) sind sehr stolz darauf, dies alles geschafft zu haben und glauben sehr daran, das

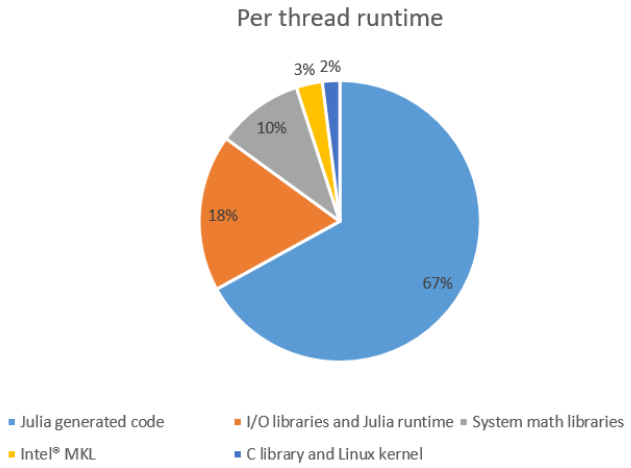

Abbildung 1: Die pro Thread Auslastung des Celeste-Projekts [Far17]

Julia die Entwicklung in der Forschung für viele Jahre sehr voranbringen wird. Viral Shah, CEO von Julia Computing, sagte dass Forscher sich nun auf die Lösung ihrer Probleme fokussieren können und sich nicht mehr vorrangig mit dem Programmieren beschäftigen müssen [JC16].

2 Einführung in die Programmiersprache Julia

2.1 Übersicht

Julia ist eine junge, flexible, höhere Programmiersprache, die es sich selbst als Ziel gesetzt hat, die hohe Produktivität und einfache Handhabbarkeit aus Sprachen wie Python mit der großen Geschwindigkeit von C zu kombinieren. Auch wenn sie vorallem für wissentschaftliches und numerisches Rechnen gestaltet worden ist, bietet sie eine sogleich performante, wie auch einfach zu lernende Umgebung und versucht damit den derzeitigen Trend von mehrsprachigen Softwarelösungen in der Wissenschaft entgegen zu wirken.
Mit einer Syntax, die an eine Kombination aus MATLAB und Python erinnert, Lisp-ähnlichen Macros und Metaprogrammierung, einem Compiler der LLVM nutzt und einem eigenem Paketmanager der an Git angelehnt ist, hat sich Julia viele bereits etablierte Funktionen zu Nutze gemacht. Zusätzlich besitzt sie die Möglichkeit C- und Python-Funktionen einfach direkt aufzurufen, was die Nutzung der Bibliotheken der beiden Sprachen direkt im Julia-Code ermöglicht. Julias gute Performanz wird vorallem durch Multimethoden für einzelne Funktionen und die automatische Generierung von effizientem, typspezialisierten Code gewonnen. Dabei ist Julia sowohl dynamisch als auch statisch typisiert. Zudem ist Julia funktional, objektorientiert und imperativ und wurde fast komplett in Julia selbst geschrieben [Wik18c] [Dan17].

2.2 Typensystem und Multimethoden

In Julia gibt es keine Klassen, wie es sonst für objektorientierte Sprachen üblich ist. Stattdessen werden in Julia Daten eines Objektes explizit von dessen Methoden getrennt. Vom Nutzer angelegte Objekte, wie sie aus anderen puren objektorientierten Sprachen wie C++ oder Java bekannt sind, ähneln am meisten dem *Composite Types*. Auch bekannt unter Namen wie *struct* oder *records* stellt er eine Ansammlung

von benannten Feldern da. Diese einzelnen Felder können wie einzelne Attribute betrachtet werden und besitzen einen Wert. Im Sinne der Objektorientierung können die *Composites* ineinander verschachtelt werden. Zusätzlich erzeugt Julia standartmäßig für jeden *Composite Type* auch Konstruktoren. Auch wenn Julia dynamisch ist und damit der Typ eines Wertes nicht angegeben werden muss, besitzt jeder einen festen Datentyp. Dies gilt auch für die vom Nutzer erstellten Typen. Datentypen existieren dabei nur zur Laufzeit und entsprechen genau dem aktuellen Typ, den ein Wert während der Ausführung besitzt. Auch besitzen nur Werte einen Typ, eine Variable stellt lediglich einen Namen gebunden an einen Datentyp dar.
Julias Rückgrat für das Typensystem sind die abstrakten Datentypen. Während ein Wert immer einen konkreten Datentyp besitzt, ist dieser Typ gleichzeitig in einen Graphen von abstrakten Typen eingebunden. Konkrete Typen sind die Knoten dieses Graphen und können keine Elternknoten sein. Abstrakte Typen dagegen können mehrere Kinder- und Elternknoten besitzen und bilden die Basis von Julia. Der Datentyp *Int64* ist z. B. Kindknoten vom abstrakten Typ *Signed*, welcher Kindknoten vom Typ *Integer* ist. Als oberster Knoten steht der Datentyp *Any*. Ein Nutzer kann seine definierten Objekte in diesen Graphen einfügen. Wird nichts weiter spezifiziert, wird das Objekt standartmäßig als Kind vom Typ *Any* angenommen.
Als ein weiteres Feature unterstützt Julia auch parameterisierte Datentypen. Diese ermöglichen die generische Erstellung von vielen kombinierten Datenypen und dienen zudem unterstützend für Julias weiteres Schlüsselmerkmal - die Multimethoden. Im Kontext von Julia stellen Funktionen ein Objekt dar, welches ein Tuple von Eingangswerten einem Rückgabewert zuordnet oder einen Fehler wirft. Eine einzelne Funktion fasst dabei einen Sachverhalt wie z. B. die Addition von Werten zusammen und besitzt unterschiedlich viele verschiedene Methoden, welche je nach Anzahl der Argumente und Typ jedes einzelnen Wertes gewählt wird. Deshalb werden Funktionen in Julia auch generisch genannt. Methoden widerrum sind nun eine explizite Implementierung einer Funktion und werden nach dem Multimethoden-Verfahren (in Englisch multiple dispatch) ausgewählt. Damit grenzt sich Julia im Sinne des objektorientierten Paradigmas sehr von Sprachen wie Java oder C++ ab. Während in diesen Sprachen eine Zuordnung nach dem Typ des ersten Arguments (single dispatch) implizit entschieden wird (z. B. etwas wie `obj.meth(arg1,arg2)`) werden Methoden in Julia explizit nach dem am exaktesten passenden Argumenttypen (z. B. etwas wie `meth(arg1::Int64,arg2::Number)`) ausgewählt. Am exaktesten passenden Typ meint hier das vom konkreten Typ des Eingabewertes aufsteigend der Typenbaum abgegangen wird, bis eine Methode gefunden wird, die der Anzahl der Argument und deren Typen entspricht. Wird keine gefunden, so wird ein Fehler geworfen.
Multimethoden kombiniert mit den Trennen der Funktionen von den Objekten, auf denen sie arbeiten und dem Typenbaum, ergibt damit eine Art Vererbung die je nach Eingabetypen einer Funktion Verhalten weiterreicht und dabei die Struktur ignoriert. Die Entwickler von Julia selbst sagen, das sie festgestellt haben, das die ledigliche Vererbung von Verhalten unter Ausgrenzung der Struktur bestimmte Schwierigkeiten der objektorientierten Sprachen lösen konnte ohne große Nachteile festzustellen. Dies ergibt sich daraus, das Mehrfachvererbung auf konkreten Typen zu einer Hierarchie führt, welche sich nicht mehr gut auf die Wirklichkeit abbilden lässt und auch allgemein alles viel mehr verkompliziert, wie man z. B. am Diamant-Problem gut erkennen kann. Julia hat sich explizit dagegen entschieden. In ihr wird jeder konkrete Typ einem seperaten Blattknoten zugeordnet und Typen mit ähnlichen Eigenschaften unter einem abstrakten Typ zusammen gefasst werden [Lan18].

2.3 Metaprogrammierung - eine Hommage an Lisp

Ein weiteres wichtiges Merkmal von Julia ist die Unterstützung für Metaprogrammierung. Wie in Lisp, repräsentiert Julia seinen eigenen Code als Datenstruktur in sich selbst. Da dieser Code durch Objekte dargestellt wird, kann er genauso aus Julia heraus auch erstellt und manipuliert werden. Dadurch wird es möglich Code zu transformieren oder gar Code zu generieren. Julia repräsentiert zudem alle Datentypen und den Code in Strukturen, welche in Julia implementiert worden sind, wodurch auch eine sehr mächtige Rückschau (engl. Reflection) möglich ist. Ein Ausdruck wird dabei immer in ein *Expr*-Objekt übersetzt, welches sowohl den Typ des Ausdrucks wie auch seine Argumente beinhaltet und in Julia sehr einfach zu bearbeiten oder auszuwerten ist. Dem Datentyp *Symbol* wird hierbei noch eine sehr bedeutende Rolle zugedacht. Er repräsentiert alle Variablen und damit verbunden auch Zuweisungen und Funktionsaufrufe. Damit ist es möglich Daten von einer Variablen auch in der Datenstruktur noch zu unterscheiden.
Julia besitzt auch Lisp-ähnliche Macros. Dies sind Funktionen, welche einen Ausdruck als Argument nehmen, ihn manipulieren oder zusätzlichen Code hinzufügen und dann einen Ausdruck zurückgeben. Der Unterschied zu herkömmlichen Funktionen ist vorallem der Zeitpunkt zu dem ein Macro ausgewertet wird. Es wird ausgeführt wenn der Code übersetzt wird und bietet damit die ideale Möglichkeit Funktionen zusätzlich zu manipulieren. Ein gängiges, später noch sichtbares Beispiel wäre hier das *@Benchmark*-Macro, welches Performanztests ausführt und am Ende neben dem Ergebnis auch die Laufzeit, die Allokationen und den benötigten Speicher für die Funktion zurückgibt.
Um die Erstellung und Manipulation von Ausdrücken zu vereinfachen gibt es zudem die Funktion, eine Variable in den Ausdruck im Nachhinein einzufügen. Dieses Verfahren nennt sich zu englisch *splicing* oder *interpolating*. Da so auch lokale Variablen in Macros übergeben werden können ist es für diese sehr nützlich, da normal übergebene lokale Variablen sonst innerhalb des Macros undefiniert sind, um das versehentliche Übernehmen oder Überschreiben von Variablen im inneren Anwendungsbereich (scope) zu verhindern [Lan18].

2.4 JIT Compiler auf Basis von LLVM

Julias Compilierungsprozess beginnt mit der Definition einer Methode und deren Ausführung. Auch wenn Julia dynamisch ist, so ist sie dennoch stark typisiert und macht sich den Low Level Virtual Machine (*LLVM*) Echtzeit (just in Time oder JIT) Compiler zu nutze, um nativen Maschinencode zur Laufzeit zu generien. Dabei besteht dieser Prozess der Generierung aus vier Stufen und für jede Stufe bietet Julia eigene Funktionen, um sich den generierten Code ausgegeben zu lassen.
In der ersten Phase, der sogenannten niedriger Code (*code_lowered*) Phase, wird der Code etwas vereinfacht und gleichzeitig deutlich ausdrucksstärker gemacht, um dem Compiler die nächsten Schritte zu vereinfachen. An dieser Stelle werden auch generische Funktionen eingebunden, welche für unterschiedliche Argumente unterschiedlichen niedrigeren Code erzeugen. In der nächsten Stufe wird versucht über alle Typen, die im Code vorkommen, eine Schlussfolgerung zu schließen und lokale Optimierungen basierend auf den ermittelten Typen zu treffen. In der dritten Phase wird dieser, an die Typen angepasste Code, dann mit dem *LLVM*-JIT-Compiler zu *LLVM*-Code übersetzt und schlussendlich in der vierten Stufe zu nativen Maschinencode herunter compiliert. Dieser Maschinencode wird dann zwischengespeichert und ausgeführt, sobald die Funktion erneut mit gleichen Argumenttypen und der gleichen Anzahl an Argumenten aufgerufen wird, wodurch sich die zeitintensive JIT- Phase ersparen lässt und eine Funktion

bei erneuter Ausführung mit gleichen Parametern sehr viel schneller läuft [Lan18] [Lan17] [Com16].

2.5 Interne und externe Bibliotheken

Auch wenn Julia noch eine sehr junge Sprache ist, bietet sie bereits jetzt eine sehr schnell wachsende Sammlung an verschiedensten Paketen, die zahlreiche Funktionen bereitstellen und sich einfach über Julias eingebauten Paketmanager verwalten lassen. Dieser installiert, deinstalliert und aktualisiert die Pakete deklarativ, das heißt er findet selbst heraus welche Versionen noch benötigt werden oder welche veraltet sind und gelöscht werden können. Intern nutzt der Manager das *libgit2* Framework und setzt damit direkt auf *git* auf. Alle Pakete sind dabei einfache *git* Repositories und klonbar, wie es von *git* bekannt ist. Die Pakete müssen dabei einer gewissen Format-Konvention folgen und im *Metadata.jl* Repository registriert sein, damit Julia das Paket finden kann, unter der Angabe der URL für unregistrierte Pakete gibt es aber eine Möglichkeit für diese, eingebunden zu werden.
Zudem wird Julias Paketmanager derzeit sehr umfangreich überarbeitet und es ist zu erwarten, das in der nächsten Version 0.7 Julias Paketverwalter um einige Funktionen erweitert und verbessert wird. Neben diesem Paketmanager für eigene Julia-Pakete bietet die Sprache aber auch eine sehr einfache Maske um C- oder Fortran-Funktionen und Werte direkt aus Julia aufzurufen und stellt damit ideale Möglichkeiten zur Integration in einer bereits etablierten Umgebung zur Verfügung. Dabei funktioniert alles ganz ohne Klebstoff-Code mit einem einfachen Funktionsaufruf wie z. B. `ccall()` und da der von Julias JIT Compiler native generierte Code am Ende der Gleiche wie der native C Code ist, gibt es keinen Verlust bei der Performanz. Auch Python oder C++ über ein Paket lassen sich sehr einfach einbinden, zudem gibt es noch die Möglichkeit aus Julia heraus direkt externe Programme aufzurufen [Lan18].

3 Parallele Programmierung mit Julia

Als Sprache, die parallele Programmierung als eine ihrer Kernkompetenzen sieht und sich sehr auf Performanz orientiert, wundert es nicht, dass Julia auch sehr viele Möglichkeiten bietet, um Berechnungen zu parallelisieren. Julia hat in diesem Feld eine Fülle von einfachen Funktionen für simple Probleme, aber gleichzeitig auch eine Tiefe der Funktionen, um noch das letzte bisschen Performanz herauszukitzeln. Als Grundbaustein für Julias Multiprozessorhandhabung dient dabei eine eigene Implementierung einer Nachrichtenübertragung zur Kommunikation zwischen den Prozessen, um diese parallel in seperaten Speicherbereichen laufen zu lassen. Diese Übertragung ist anders als bestehende Konzepte wie MPI und ist so ausgerichtet, dass generell nur eine Seite programmiert werden muss, um eine Kommunikation aufzubauen und zu beenden und zusätzlich auch eher wie ein Funktionsaufruf aussieht.
Um eine Unterscheidung zwischen echten Kernen eines Rechnersystems und den eingebundenen Kernen für die Berechnungen in Julia treffen zu können, wird im Folgenden mit dem Wort *Prozessoren* immer die Anzahl der Kerne in der Julia-Anwendung bezeichnet. In Julia besitzt jeder Prozessoren eine eindeutige Kennzeichnung. Sie macht sich zudem das *Master-Worker*-Prinzip zunutze und ernennt den Hauptprozessor zum *Master*-Prozessor mit der Id eins. Solange es mehr als einen Prozessor gibt, ist der Hauptprozess mit der Id gleich eins immer der *Master*, ansonsten wird er allein als *Worker* angenommen. Dadurch ergibt sich auch, dass das Hinzufügen nur eines weiteren Prozessors kaum Performanzgewinn erzielen kann, da der Meisterprozessor bei parallelen Aufgaben nur verwaltet und selber nicht berechnet, außer man weist ihn dazu explizit an.

Intialisiert werden können weitere Prozessoren zum einen lokal beim Starten einer Julia-Instanz mit dem Attributen `julia -p n`, wobei *n* für die Anzahl der *Worker* steht, oder aber mit der Funktion `addprocs(n)` sowohl lokal, wie auch remote, wobei hier auch zu der externen Machine via *ssh* eine Verbindung aufgebaut werden muss. Mit der Funktion `workers()` oder `nworkers()` wird ein Array mit den Ids der eingebundenen Arbeiter beziehungsweise die Anzahl der Arbeiter zurückgegeben. Die Funktion `rmprocs(id)` entfernt hingegen nur den *Worker* mit der spezifizierten Id [Lan18] [Wel18].

3.1 Remote-Aufrufe und -Referenzen

Julias parallele Programmierung ist auf zwei primitiven Objekten aufgebaut: Remote-Referenzen und Remote-Aufrufe.
Eine Remote-Referenz ist eine Instanz, die von jedem Prozessor aus genutzt werden kann und auf ein auf einem bestimmten Prozessor gespeicherten Objekt zeigt. Ein Remote-Aufruf ist ein Antrag von einem Prozessor eine bestimmte Funktion mit bestimmten Argumenten auf einem Prozessor, der auch der gleiche wie der aufrufende sein kann, auszuführen. Remote-Referenzen gibt es in der Ausprägung *Future* und als einen *RemoteChannel*. Ein *Future* ist der Rückgabewert eines Remote-Aufrufs, der direkt zurückgegeben wird, um den Prozess während seiner Ausführung nicht zu blockieren. Dieser kann dan mittels eines `fetch()` geholt werden, sobald das Ergebnis vorliegt. Ein wichtiger Hinweis ist hier zudem, das ein einmal mit `fetch()` geholtes *Future* lokal gespeichert wird und deshalb bei einem erneuten `fetch()` keine weitere Kommunikation anfällt.
RemoteChannel wiederum sind wiederbeschreibare Kanäle, die auch als Überbrücker zwischen Kanälen (Channel) verstanden werden können. Ein Kanal ist dabei eine Leitung zwischen Prozessen (Tasks) oder eben aber auch Prozessoren die genutzt werden um Daten auszutauschen, wobei jeweils immer einer Daten auf den Kanal legt und einer diese Daten abgreift, sobald sie verfügbar sind via `put!()` und `take!()`. Ein Kanal kann dabei immer mehrere schreibende wie auch mehrere lesende Prozesse haben. Ein *RemoteChannel* nimmt hier nun den Aus- oder Eingang eines Kanals auf einem vorher spezifizierten Prozessor und schickt dann die Daten weiter. Julia bietet auch zwei Funktionen an, um eine Remote-Aufruf direkt auszulösen. Normalerweise sind dies aber eher tiefer liegende Funktionen, welche der Nutzer nicht selber benutzen muss, sondern die generiert werden durch höhere Funktionsaufrufe. `remotecall(f,id,args..)` gibt dabei direkt ein *Future* zurück, während `remotecall\_fetch(f,id,args` auf den Rückgabewert wartet und darauf dann direkt noch ein *Fetch* ausführt [Lan18].

3.2 @Spawnat und @Spawn

Julia bietet neben dem Remote-Aufruf auch noch zwei weitere Macros an, um einen direkten Aufruf an einen anderen Prozessor zu starten und dafür ein *Future*-Referenz zu bekommen. Das `@spawnat`-Macro akzeptiert hierbei zwei Argumente: *pid* und *expr*. Es berechnet den Ausdruck des zweiten Arguments auf dem Prozessor spezifiziert durch das erste Argument. Damit verhält sich das `@spawnat` Macro sehr ähnlich zu einem `remotecall()`, arbeitet dabei jedoch mit einem Ausdruck statt mit einer Funktion.
Um diesen Sachverhalt noch zu vereinfachen, bietet Julia außerdem das `@spawn`-Macro an. Dieses sucht selbständig nach einem unbeschäftigten Prozessor und lagert die Berechnung der Funktion an diesen aus. Dabei verhält sich `@spawn` auch intelligent und erkennt, wenn Werte vorher auf einem bestimmten Prozessor ausgerechnet worden waren, und lagert die darauf aufbauende Berechnung auf

eben jenen Prozessor erneut aus um sich den sonst nötigen Fetch zu ersparen. Wichtig zu beachten bei einem Remote-Aufruf ist, das die aufzurufenden Funktionen dem Prozessor bekannt sind, der sie ausführen soll, gleiches gilt auch für geladene Pakete und deren Funktionen und Variablen. Mit dem Macro `@everywhere` kann eine Funktion direkt auf allen Prozessoren, definiert werden um dieses Problem zu umgehen. Auch sollte beim Umgang mit einem direkten Remote-Aufruf vorsichtig mit lokalen Variablen umgegangen werden, um einen unnötigen Kommunikationsaufwand zu vermeiden. So würde

```
julia> message = "This string is constructed locally";
julia> shouting_message = @spawn uppercase(message)
```

das Kopieren der Nachricht vom *Master*-Prozessor auf den berechnenden Prozessor erfordern und deutlich mehr Zeit kosten, als die lokale Variable direkt im `@spawn` Befehl zu definieren mit

```
julia> shouting_message = @spawn uppercase("This string is not constructed locally")
```

Aufgepasst werden sollte zudem bei globalen Variablen, da diese nicht synchronisiert werden und sich nicht durch Berechnungen auf anderen Prozessoren beeinflussen lassen [Lan18].

3.3 Parallele Map und For-Schleife

Der obere Teil der Ausführung über die parallele Programmierung war eine kurze Einführung in Julias Handhabung von parallelen Berechnungen im Hintergrund. Aber nur in den seltensten Fällen wird ein Entwickler in Julia wirklich in die Lage kommen, eine Berechnung manuell auszulagern. Stattdessen bietet Julia zwei Funktionalitäten an, um Probleme sehr einfach und gut lesbar zu parallelisieren.
Die erste Variante umfasst hierbei das Macro `@parallel`, eine parallele For-Schleife, welche besonders optimiert ist für Probleme mit vielen voneinander unabhängigen Iterationen. Dabei ist diese For-Schleife gleichzeitig auch noch eine Reduktionsfunktion und bietet die Möglichkeit eine Funktion zu übergeben, welche am Ende über allen Resultaten ausgeführt wird und dabei die Dimension des Problems um eins reduziert (auch als *tensor-rank-reducing* bekannt). Diese Form ist aber optional und eine parallele For-Schleife funktioniert auch ohne diese übergebene Funktion und gibt für diesen Fall einfach das Ergebnis zurück. Die Berechnung, wie (n-mal) häufig Kopf bei M Münzwürfen auftrat, lässt sich beispielsweise sehr einfach Parallelisieren und nutzt zudem die Addition als Reduktion:

```
julia> nheads = @parallel (+) for i = 1:M; Int(rand(Bool));end
```

Wichtig zu beachten ist hierbei dass, auch wenn sie einer normalen For-Schleife sehr ähnlich sieht, sich eine parallele For-Schleife anders verhält und die Iterationen nicht in Reihenfolge ablaufen und auch globale Variablen sich so nicht beschreiben lassen. Ein weiterer wichtiger Hinweis an dieser Stelle ist, dass der Rückgabewert einer solchen parallelen For-Schleife ohne Reduktionsfunktion ein Array von *Future*'s ist und die Funktion ohne ein `fetch()` direkt weiter läuft, auch wenn am Ende der Schleife noch kein Ergebnis vorliegt. Hilfreich hierfür ist das Macro `sync`, welches vor das Macro `parallel` geschrieben wird, um den Prozess zu blockieren, solange noch Berechnungen ausgeführt werden.
Als zweites Werkzeug stellt Julia eine parallele Map bereit, welche besonders nützlich für große Funktionen ist, die einzeln auf eine Menge von Elementen angewandt werden müssen. Die Funktion `pmap(i-> log(i), 1:I`

berechnet zum Beispiel den Logarithmus der Zahlen eins bis M und speichert sie erneut in einem Array ab. Da `pmap()` vorallem für größere Funktionsaufrufe gestaltet worden ist, kann es hier teils extreme Performanzeinbuse bei dem Aufruf von kleinen Berechnungsfunktionen wie `log()` geben, da der Kommunikationsaufwand sehr groß ist, wenn jeder Wert einzeln auf einem neuen Prozessor berechnet wird und dafür jeweils immer der Wert hin und her kopiert wird. Eine Lösung bietet die Funktion dafür aber direkt selber. Mit dem Attribut *batch_size* kann die zu kopierende Datengröße pro Funktionsaufruf für einen Prozessor festgelegt werden. Die Nutzung von `@parallel`, das für kleine Iterationen optimiert ist, wäre hier aber genauso möglich [Lan18].

3.4 Shared und Distributed Arrays

Shared Arrays nutzen geteilten Speicher im System um das gleiche Array über viele Prozessoren hinweg zu verlinken. Im Gegensatz zu einem *Distributed Array*, in dem jeder Prozess seinen eigenen lokalen Zugang zu einem eigenen Teil (*chunk*) des gesamten Arrays besitzt, hat jeder teilnehmende Prozess bei einem *Shared Array* Zugang zum gesamten Feld. Während *Distributed Arrays* vorallem für sehr große Felder, welche zu groß zum Rechnen auf einem einzelnen Prozessor sind, geeignet sind und bei diesen jeder Prozess seinen Teil der Daten verarbeitet, eignen sich *Shared Arrays* besonders für gemeinsames Rechnen, in dem darüber der Kommunikationsaufwand deutlich reduziert wird und zum Beispiel auch Iterationen über globale Arrays mit der parallelen For-Schleife möglich werden. Mit den Funktionen `sdata()` sowie `convert()` und `localpart()` für *Distributed Arrays* lassen sich beide Arrays ganz einfach zu normalen Standartfeldern konvertieren beziehungsweise den lokalen Anteil des *Distributed Arrays* in ein normales Array extrahieren [Lan18].

3.5 Softwarewerkzeuge

Um das Erlernen der Sprache Julia zu erleichtern oder auch um einfach Fehler und schlecht funktionierende Codezeilen aufzuzeigen, gibt es bereits einige sehr praktische Werkzeuge für Julia. BenchmarkTools zum Beispiel ist ein Paket, welches geschrieben wurde, um mathematisch korrekte Maßstabanalysen (Benchmarks) zu gewährleisten. Neben der Möglichkeit bestimmte Handlungen, wie das Erzeugen von Zufallszahlen auszuklammern, um den Test aussagekräftiger zu gestalten, bietet es zudem verschiedenste Macros und Funktionen an, um die Allokationen, den Speicherverbrauch und auch die Laufzeit der Funktionen zu messen und gibt dabei sogar Minimale, Maximale und Durchschnittswerte zurück.
Ein weiteres sehr hilfreiches Werkzeug sind Julias eigene Macros `@code_native`, `@code_typed`, `@code_llvm` und vorallem `@code_warntype`, welche den compilierten Code für die entspreche Funktion in dem gewählten Level zurück geben und damit unerwartetes Verhalten gut aufzeigen können. `@code_warntype` hebt dabei sogar alle ungenauen Typ-Spezifikationen farblich hervor und eignet sich damit ideal, um zu überprüfen, ob jede Variable den günstigsten Typ zugewiesen bekommen hat. Denn auch wenn Julia dynamisch ist und Typen nicht explizit genannt werden müssen, kann man so den Code deutlich beschleunigen, da Checks entfallen oder Berechnungen aufgrund gewisser Typinformationen deutlich vereinfacht werden können.
Ein weiteres Instrument bietet Julia hier auch noch mit dem Profiling, welches ermittelt, wieviel Zeit pro indivdueller Zeile benötigt worden ist. Auch gibt es einen Linter, welcher Hinweise zum Code gibt

und damit aufzeigt, wenn einfachere Implementierungen möglich sind. Julias Debugger ist in seiner derzeitigen Implementierung nicht kompatibel mit der aktuellen Version von Julia und kann deshalb auch kaum benutzt werden, da es z. B. keine Breakpoints gibt [Lan18].

3.6 Vertiefende Themen zum parallelen Programmieren

Neben diesen doch recht einfachen anwendbaren Funktionalitäten bietet Julia auch noch eine Reihe weiterer Möglichkeiten zusätzlichen Performanzgewinn zu erzielen. Die erste Möglichkeit ist dabei die manuelle Handhabung der Lebenszeit der Objekte. Auch wenn der implementierte Garbage-Collector selbst mit der Zeit Objekte aufräumt, so sollte gerade bei Codezeilen mit sehr vielen kleinen und auch kurzlebigen Objekten explizit die Funktion `finalize()` aufgerufen werden, um den Speicher von lokalen Instanzen, Remote-Referenzen oder auch Shared Arrays freizugeben. Der Remote-Knoten, von dem diese Objekte referenziert werden, kann sonst sehr groß werden und trotzdem vom Garbage Collector übergangen werden, da dieser größere Objektereferenzen zuerst entfernt und damit die vielen kleinen Referenzen des Remote-Knotens unbeachtet lässt. Ein Löschen des mit `fetch()` geladenen Future's ist nicht notwendig, da `fetch()` bereits den Remote-Knoten entfernt.
Eine weitere Möglichkeit bietet die eigene Konfiguration eines Clustermanagers. Dieser ist verantwortlich für das Starten der Arbeiter, das Verwalten von Events, die auf den Arbeitern auftreten, und auch für den Datentransport und der Kommunikation zwischen den Prozessoren. Dabei kann beispielsweise vom standartmäßigen TCP/IP hin zu MPI abgewichen oder verschiedene Aufgaben-Warteschlangen wie Slurm oder PBS realisiert werden, um eine effizientere Auslastung der Prozessoren zu erreichen.
Auch stellt Julia mit dem Macro `@simd` die Option, dem Compiler explizit mitzuteilen, an bestimmten Stellen zu vektorisieren. Zusätzlich gibt es auch noch verschiedenste Pakete wie z. B. das ParallelAccelerator-Paket von Intel [Lan16], welches mit dem `@acc`-Macro ein Mittel bietet, bestimmte Teile wie in diesem Fall ressourcenkostende Arrayfunktionen noch einmal deutlich in der parallelen Ausführzeit zu beschleunigen. Dies gelingt, indem mit `@acc` annotierter Code erst optimiert und dann zu C++ OpenMP Code compiliert wird.
Daneben gibt es auch noch experimentielle Möglichkeiten, welche sich derzeit noch in Entwicklung befinden, aber bereits eingeschränkt nutzbar sind. So lässt sich die Netzwerktopologie des Clusternetzwerkes in die Funktion `addprocs()` als Argument mitgeben, um zu spezifizieren, wie der Arbeiter zu den anderen Prozessoren verbunden werden soll. Auch gibt es bereits erste Möglichkeiten für Multithreading in Julia, wobei hier vorher noch eine Umgebungsvariable außerhalb von Julia gesetzt werden muss um festzulegen, wieviele Threads gestartet werden sollen. Mit dem Macro `Threads.@threads` lassen sich dann Schleifen parallelisieren, ähnlich zur `@parallel`-Variante nur ohne Reduktionsfunktion [Lan18].

4 Bewertung der HPC-Eignung von Julia

Julia erhebt für sich selbst den Anspruch eine sehr performante Sprache zu sein und hat demzufolge auch Ansprüche als Programmiersprache für HPC-Systeme eingesetzt zu werden. Um dieses Vorhaben beurteilen zu können, wurden eine Reihe von Benchmark-Tests durchgeführt. Dabei wurde die sequenzielle Performanz im Vergleich zu C und Python, die Durchschnittslaufzeit im Vergleich zur Maximum- und Minimumlaufzeit für sequenzielle Probleme und die parallele Performanz für verschiedene Implemen-

tierungen eines sehr gut zu parallelisierenden Problems und eines mittelmäßigen Problems betrachtet. Als Vergleichkriterium diente dabei immer die Laufzeit der einzelnen Funktionen, Zufälligkeiten, Unregelmäßigkeiten und Seiteneffekte wurden versucht so klein wie nur möglich gehalten zu werden.
Für die sequenziellen Tests wurden die offiziellen Benchmarks der Julia-Lang-Community [Lan14] benutzt und sind als solche auch mit dementsprechender Vorsicht zu betrachten, da Probleme unter Umständen so gewählt wurden, dass Julia besonders gut aussieht. Nicht desto trotz stellen sie aber ein Vergleichskriterium dar und wurden schlussendlich vorallem gewählt, weil sie nicht die bestmögliche Implementierung in der jeweiligen Sprache darstellen, sondern so ähnlich wie nur möglich geschrieben wurden.
Alle Ergebnisse für die sequenziellen Laufzeittests und der parallelen Tests wurden auf dem Taurus-System der Technischen Universität Dresden, explizit einem Haswell-Knoten mit 2x Intel(R) Xeon(R) CPU E5-2680 v3 (je 12 Kerne) @ 2,50 GHz, Multithreading ausgeschalten, 128 GigaByte SSD als lokale Festplatte und exklusiv Rechten berechnet.

4.1 Laufzeittests gegen C und Python für sequenzielle Probleme

Wie bereits erwähnt, wurden für die folgenden Vergleiche von Julia mit anderen Programmiersprachen öffentliche Microbenchmarks von der Julia-Lang-Community [Lan14] benutzt. Dabei wurden lediglich Julia, C und Python miteinander verglichen. Julia trat hierbei mit der Version 0.6.2 gegen den 7.1 gcc-Compiler und den Python-Interpreter Version 3.6.0 an. Als Benchmarks wurden verschiedene typische Programmmuster getestet, wie etwa Funktionsaufrufe, Zeichenkettenzergliederung, Sortieren von Daten, nummerische Schleifen, Zufallszahlengenerierung, Rekursion und Operationen auf Feldern. Alle Benchmarks wurden laut Julia-Lang so geschrieben, dass sie zwar nicht die maximale Performanz der Sprache erreichen, wohl aber identische Algorithmen nutzen und damit vorrallem zeigen, wie performant selbstgeschriebener, weniger optimierter Code in C, Julia und Python ist. Zum Beispiel nutzt der `iteration_pi_sum` Algorithmus für jede Programmiersprache die gleiche For-Schleife und sieht zudem fast komplett identisch zwischen Python und Julia aus (siehe Abbildung 2).

```
function pisum()
    sum = 0.0
    for j = 1:500
        sum = 0.0
        for k = 1:10000
            sum += 1.0/(k*k)
        end
    end
    sum
end
```

```
double pisum() {
    double sum = 0.0;
    for (int j=0; j<500; ++j) {
        sum = 0.0;
        for (int k=1; k<=10000; ++k) {
            sum += 1.0/(k*k);
        }
    }
    return sum;
}
```

```
def pisum():
    sum = 0.0
    for j in range(1, 501):
        sum = 0.0
        for k in range(1, 10001):
            sum += 1.0/(k*k)
    return sum
```

Abbildung 2: Berechnung der Reihesumme $\sum_{q=0}^{10000} \frac{1}{q^2}$ 500 - mal in Julia, C und Python v. l. n. r.

Wie in Abbildung 3 zu sehen, ist hierbei Python teils deutlich langsamer und verlangte sogar eine Eingrenzung der Skale, da es bei `recursion_fibonacci` mehr als das 100-fache der Zeit, die C für den gleichen Algorithmus benötigt hatte, brauchte. So fällt neben diesem deutlich schlechterem Abschneiden von Python vorallem auf, dass Julia etwa ähnlich schnell wie C, bisweilen sogar schneller als C lief, was aber eindeutig den Benchmarks und dessen verminderter Aussagekraft an dieser Stelle zuzuschreiben ist. Allerdings ist z. B. auch an der Stelle für die Matrix-Matrix-Multiplikation erkennbar, wie schnell

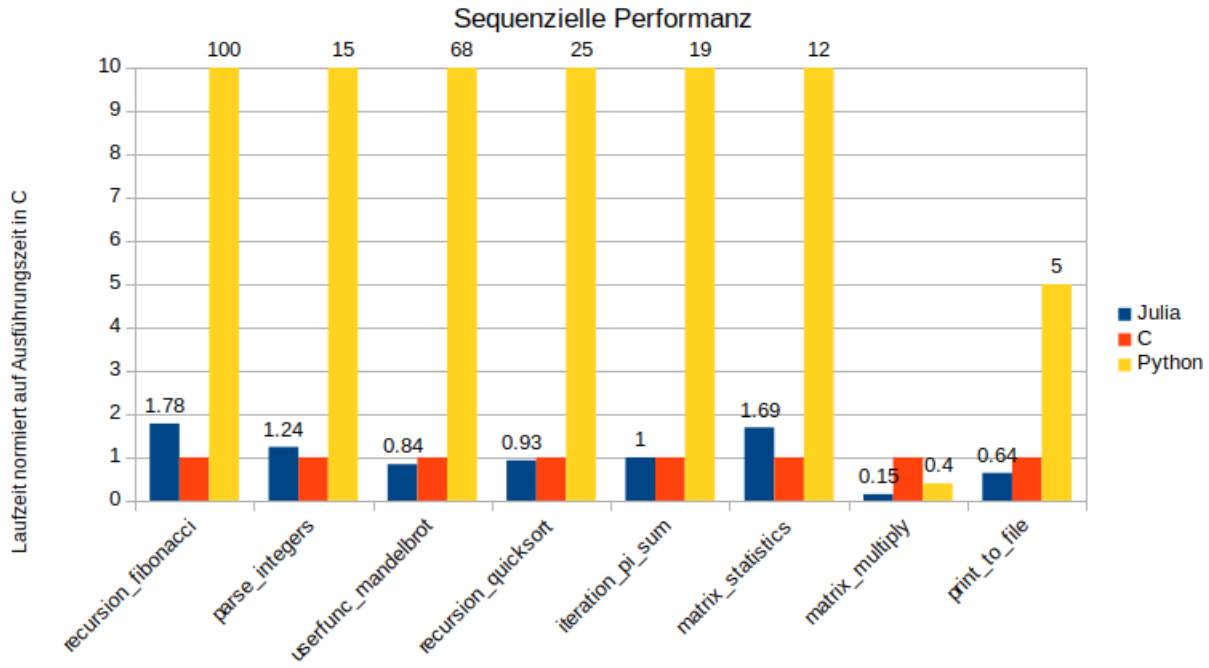

Abbildung 3: Sequenzielle Performanz von Julia, C und Python für verschiedene Benchmarks

Julias Code werden kann, wenn der Compiler in diesem Fall beispielweise zusätzliche Informationen über die Art der Matrizen erhält. Julia stellt damit vorallem unter Beweis, Python als Sprache für das Rapid-Prototyping ablösen zu können, da es eine ähnlichen angenehmen Syntax bietet, mit diesem aber eine viel bessere Laufzeit erreichen kann.
Ein wichtiger Augenmerk sollte hier aber noch darauf gelegt werden, dass Julia JIT compiliert. Dadurch kann der erste Aufruf einer Funktion wesentlich länger dauern, als folgende Aufrufe mit gleicher Argumentanzahl und Argumenttypen (siehe Abbildung 4). Nach diesem Erstdurchlauf geht die Geschwindigkeit aber aufgrund des zwischengespeicherten Machinencodes deutlich nach oben und ermöglicht Julia damit so viel schneller als Python zu sein.

4.2 Laufzeittests für parallele Performanz in Julia

Neben diesen Microbenchmarks wurden weiterhin Benchmarks für die oben aufgeführten parallelen Verfahren durchgeführt und gegen die sequenzielle Performanz gemittelt, um auch eine Aussagekraft über Julias Eignung auf einem HPC-System und den dort vorliegenden Anforderungen vorbringen zu können. Die Laufzeittests wurden mithilfe des Programms, veröffentlicht auf [Wel18], ermittelt. Betrachtet wurden hierbei die sequenzielle Performanz, die Performanz von `@parallel`, das `@parallel`-Macro in Kombination mit der Reduktionsfunktion im Speziellen die Addition `(+)`, sowie `pmap()` mit und ohne der Konfiguration von der Batchsize-Größe. Die folgendene Abbildung 5 zeigt ein Code-Snippet der genutzten Verfahren in ihrer Implementierung.
Alle Verfahren wurden benutzt, um die Summe über einem Array bzw. einem Shared-Array mit vorher festgelegter Länge zu berechnen. In den einzelnen Feldern des Arrays wurden vorher Werte durch eine Summenfunktion `fun` über der Wurzel ausgerechnet und zurück geschrieben. Dabei werden die Laufzeittests in zwei Kategorien unterteilt: gute und mittlere Parallelisierbarkeit. Gute Parallelisierbarkeit

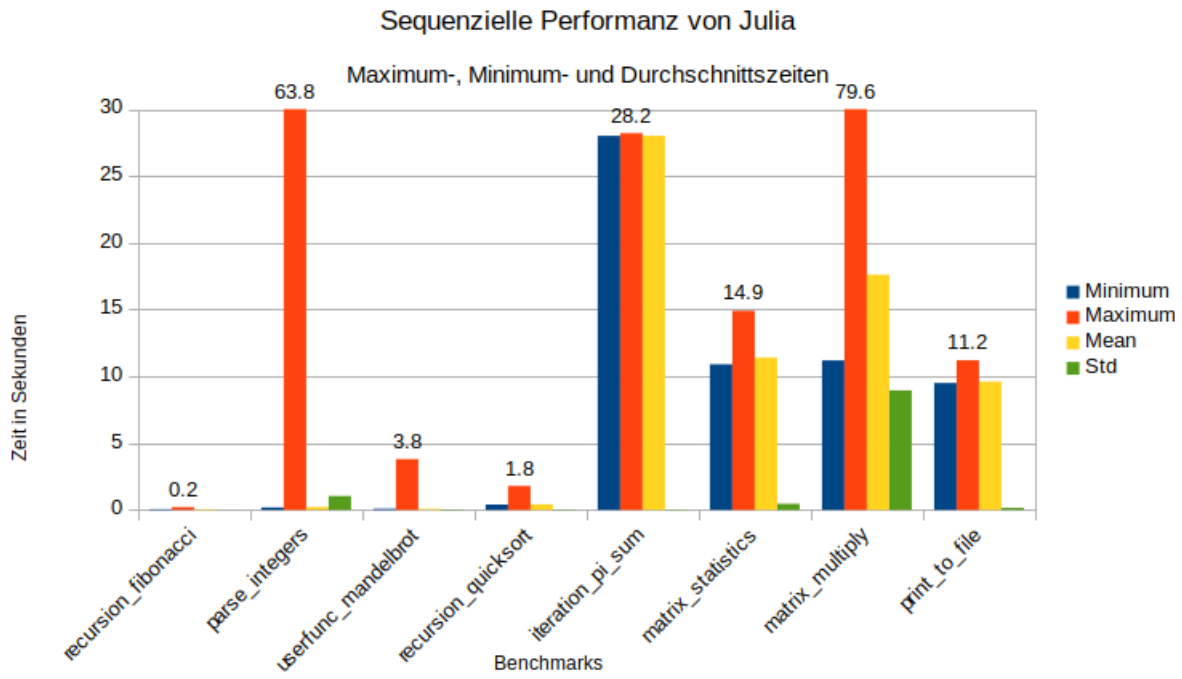

Abbildung 4: Minimum-, Maximum-, Mean- und Standartwerte von Julia für die sequenzielle Performanz der verschiedenen Benchmarks

bezeichnet hier wenige Funktionsaufrufe mit schwereren, aber weniger speicherverbrauchenden Funktionsberechnungen und mittlere Parallelisierbarkeit steht für mehr Funktionsaufrufe mit mittleren, mehr speicherverbrauchenden Funktionen. Dies wurde über die Parameter `ProblemSize` und `FunCalls` global konfiguriert, indem diese direkt den Aufwand für die Funktion `fun` und die Größe der Arrays und damit auch der Funktionsaufrufe festlegten. Gut erkennbar ist zudem, wie ähnlich sich die einzelnen Implementierungen sehen.

Abbildung 6 zeigt die berechneten Werte für die verschiedenen Verfahren bei guter Parallelisierbarkeit in Abhängigkeit von der Anzahl der in Julia eingebundenen Prozessoren und dem Speedup, normiert auf die sequenziellen Performanz. Gut zu erkennen ist, das sich für weniger, aber dafür größere Funktionsaufrufe die parallele Map gegen die For-Schleife durchsetzt und nochmals einen doppelten Speedup im vergleich zu dieser erzielen kann. Trotzdem ist auch die parallele For-Schleife um den Faktor fünf bis sechs schneller als die sequenzielle Performanz. Auffällig ist zudem das Sinken der Performanz beim Hinzufügen nur eines weiteren Prozessors, was sich durch die Implementierung des *Master-Worker*-Prinzips in Julia erklären lässt. Da der erste Prozessor zum *Master*-Prozessor wird, berechnet dieser nichts und die komplette Berechnung findet sequenziell auf dem Prozessor mit der Id zwei statt. Die Kosten für die Kommunikation und der Overhead, um den Prozess parallel zu machen, fallen aber trotzdem an, weshalb die Zeit, die der Algorithmus am Ende benötigt, sogar noch steigt. Das Festlegen der zu kopierenden Größe (Batchsize) für die parallele Map sorgt zudem für eine leicht bessere Performanz und verhindert, das die Kommunikation für ungünstige Situationen, wie etwa für zwei Prozessoren, den Algorithmus stark verlangsamt. Auch die Reduktionsfunktion der For-Schleife ist leicht schneller, was vorallem dem in diesen Messwerten nicht beachtetem Speichermanagment zuzuschreiben ist. Da die Reduktionsfunktion den Speicher effizienter verwaltet, kann sie schneller sein, belegt während der Ausführung vorallem aber auch weniger Speicher.

```
function simple_loop_sum(pt::Int,fun::Function=GlobalFun)
    a= Array{Float64}(FunCalls)
    for i=1:FunCalls;
        a[i] = fun(ProblemSize);
        end#for
    sum(a)
end#function##

function sharedarray_parallel_sum(pt::Int,fun::Function=GlobalFun)
    sa = SharedArray{Float64}(FunCalls)
    s= @sync @parallel for i=1:FunCalls;
        sa[i] = fun(ProblemSize);
         end#for
    sum(sa)
end#function

function sharedarray_mapreduce(pt::Int,fun::Function=GlobalFun)
    sa=SharedArray{Float64}(FunCalls)

    @parallel (+) for i=1:FunCalls;
        sa[i]= fun(ProblemSize);
        end#for
    sum(sa)
end#function

function pmap_sum_nb(pt::Int,fun::Function=GlobalFun)
    r = pmap( i-> fun(ProblemSize), 1:FunCalls )
    sum(r)
end#function

function pmap_sum_b(pt::Int,fun::Function=GlobalFun)
    r = pmap( i-> fun(ProblemSize), 1:FunCalls,
        batch_size=ceil(Int,FunCalls/nworkers()))
    sum(r)
end#function
```

Abbildung 5: Code-Snippets der benutzten Verfahren mit *ProblemSize* als Problemgröße und *FunCalls* als Anzahl an Funktionsaufrufen

In der Abbildung 7 sind die gemessenen Werte der einzelnen Verfahren bei mittlerer Parallelisierbarkeit wieder in Abhängigkeit der oben genannten Größen erkennbar. Auffällig ist die sehr schlechte Performanz der `Pmap()`-Funktion ohne Batchsize. Mit bis zu 23-fachem Zeitaufwand verglichen mit der sequenziellen Performanz ist dieses Verfahren ohne das Spezifizieren der Batchsize nicht mehr rentabel und verschlimmert sogar die Performanz des Algorithmus deutlich. Deshalb wird dieses Verfahren im Weiteren nur noch mit einer festgelegten Batchsize betrachtet. Aber auch so ist die parallele Map nur noch um den Faktor zwei schneller als die sequenzielle Performanz und verliert damit deutlich auf die parallele For-Schleife im Vergleich zur Abbildung 6. Gut zu erkennen ist hierbei, das jedes Verfahren für seine Probleme optimiert ist und `Pmap()` eher für größere Funktionen und `@parallel` eher für viele kleinere Funktionen geeignet ist. Die parallele For-Schleife wird für mittlere Parallelisierbarkeit teilweise sogar noch schneller als für gute Parallelisierbarkeit und erreicht wieder einen Speedup ca. um den Faktor fünf bis sechs. Auffällig ist auch, dass der Algorithmus sich nicht durch das Hinzufügen von weiteren Prozessoren unbeschränkt beschleunigen lässt, sondern ab 14 eingebundenen Prozessoren durch Kontextwechsel und weiteren Kommunikationsaufwand anfängt etwas langsamer zu werden. Auch der bereits beobachtete Abfall der Performanz für zwei eingebundene Prozessoren lässt sich hier wieder beobachten und mit der gleichen Argumentation wie oben erklären.

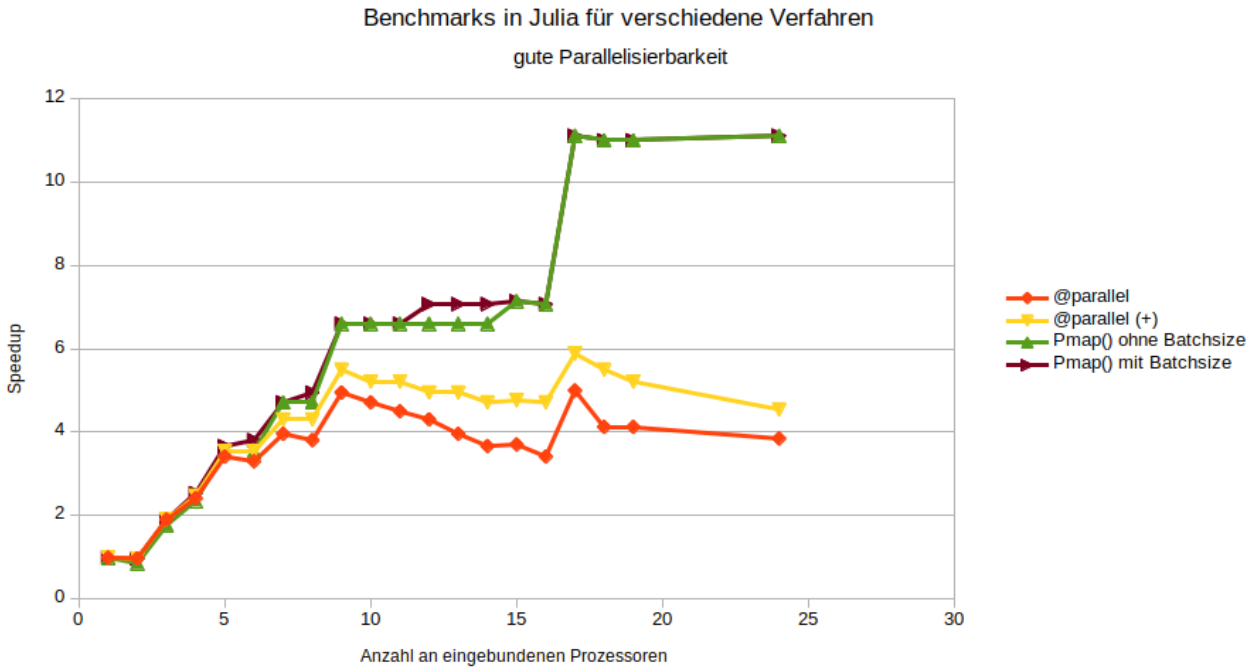

Abbildung 6: Graph der Messwerte für gute Parallelisierbarkeit auf Taurus berechnet

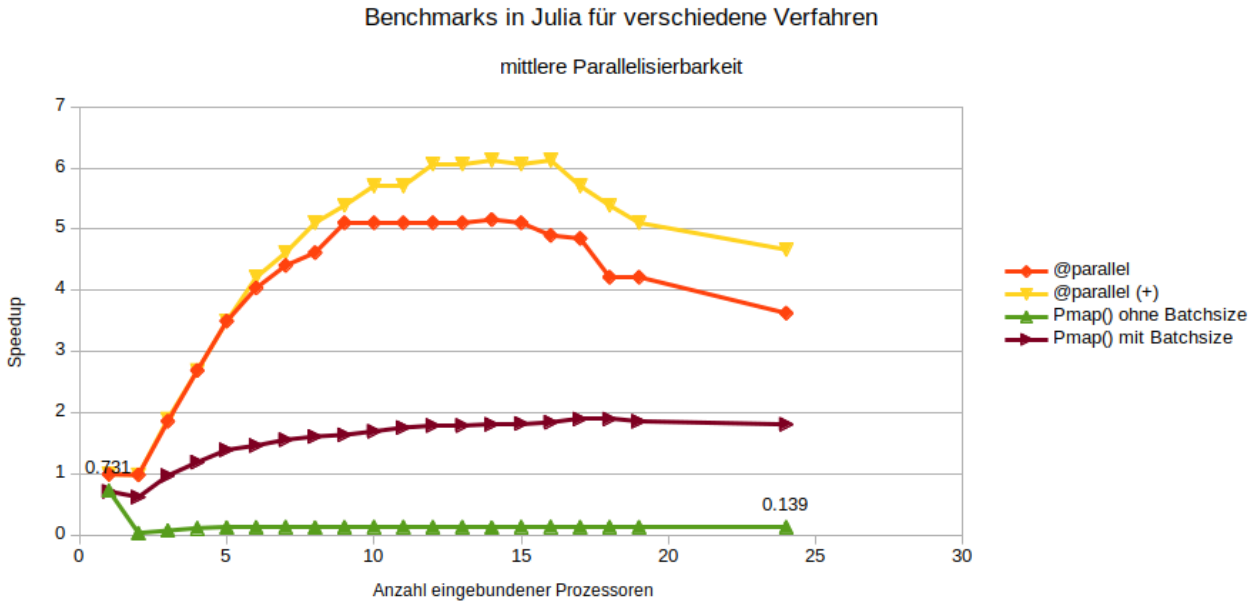

Abbildung 7: Graphen der Messwerte für mittlere Parallelisierbarkeit auf Taurus berechnet

5 Zusammenfassung

In dieser Ausarbeitung wurden Julia hinsichtlich der Kriterien Handhabbarkeit und Performanz für die Programmierung von HPC-Systemen analysiert. Dazu wurden im ersten Teil Julias Schlüsselfunktionen betrachtet und darauf untersucht, wie sich Julia deren Vorteile Zunutze macht und gleichzeitg deren Schwachstellen ausgleicht etwa durch das Speichern von Maschinencode. Im zweiten Teil wurde dann Bezug zur High-Performance-Programmierung hergestellt, indem sich ausführlich mit Möglichkeiten zur Parallelisierung und Geschwindigkeitserhöhung auch unter Betrachtung von Fallstricken in Julia beschäftigt wurde. Dabei wurde zuerst die Funktionen im Hintergrund von Julia thematisiert, die die Parallelität gewährleisten. Im Anschluss wurden die gängisten Verfahren zur parallelen Programmierung mit Julia vorgestellt und zum Schluss noch etwas über weiterführende Themen und hilfreiche Werkzeuge gesagt. Im dritten und abschließenden Teil wurden dann die Eignung von Julia als High-Performanz-Sprache untersucht. Hierfür wurden sequenzielle Tests gegen C und Python durchgeführt und ausgewertet und anschließend noch oben angesprochenen Verfahren hinsichtlich ihrer Performanz für unterschiedlich gut zu parallelisierenden Fällen betrachtet.
Julia kann eine sehr praktische und auch schnelle Sprache sein. Viele der getesteten Algorithmen konnte Julia teils deutlich beschleunigen ohne dabei den angenehmen Syntax von Python aufgeben zu müssen. Die Kombination aus typoptimierten Code mit ei6+6nem Compiler, dem man Details mitgeben kann, wie etwa die innere Struktur einer Matrix, sorgt für sehr gute sequenzielle Geschwindigkeiten, sofern die Funktionen vorher bereits einmal ausgeführt worden waren und fast ausschließlich gespeicherter Maschinencode benutzt werden kann. Auch für High-Performance-Programmierung bietet Julia viele angenehme Funktionen, die Algorithmen in ihrer Ausführung stark beschleunigen, sofern Fallstricke und optimale Möglichkeiten beachtet werden. Julia kann, wie im Projekt Celeste [Far17] beschrieben, einen sehr großen Speedup erreichen und sich sogar fast mit der Geschwindigkeit von C messen lassen und kann damit zurecht behaupten, Python als Rapid-Prototyping-Sprache zu verdrängen und durch kleine Optimierungen und Analysen des Julia-Codes mit Werkzeugen auf die Auslagerung in C-Routinen verzichten zu können. Auch bestehende Bibliotheken aus anderen Sprachen lassen sich sehr einfach einbinden und ausführen und erreichen dabei zumeist noch ähnliche Geschwindigkeiten, wie in der ursprünglichen Sprache. Auch die Communty ist sehr aktiv und hilfsbereit, wodurch sich der Einstieg in diese noch sehr junge Sprache deutlich vereinfachen lässt. Ein großer Nachteil den Julia allerdings immer noch mit sich bringt und der dafür sorgt, das sich die Sprache bisher noch nicht etablieren konnte, ist dessen Instabilität und häufigen Patches aufgrund der Beta-Version in der sie sich aktuell noch befindet. Selbst eigentlich aktuelle Pakete, wie die Microbenchmarks, erfordern Anpassungen im Quellcode um ausgeführt werden zu können. Mit im Zusammenhang dazu steht auch die Softwareumgebung u. a. die vielen Pakete und Tools, die es bereits gibt, die aber wie z. B. der Debugger teilweise einfach noch nicht mit der aktuellen Version im vollem Umfang funktionieren und dadurch gerade die Entwicklung von Anwendungen, die nicht auf numerisches Rechnen ausgelegt sind, erschweren und notwendig machen, sich selbst mit den Paketen und deren Implementierung zu befassen.
Trotz alledem ist Julia eine interessante neue Sprache, die es durch aus Wert ist, weiter untersucht zu werden und sich zurecht als High-Performanz-Programmiersprache bezeichnet. Gerade nach dem ersten öffentlichen Release sollte sie als Konkurrent zu Python und C betrachtet werden und als Programmiersprache für ein Projekt zumindest überdacht werden.

Literatur

[Ast14] ASTRONEWS.COM: *SDSS III: Entfernte Galaxien genau vermessen*. http://www.astronews.com/news/artikel/2014/01/1401-013.shtml, 2014. – [Online; accessed 22-May-2018]

[Com16] COMPUTING, Julia: *Static and Ahead of Time (AOT) Compiled Julia*. https://juliacomputing.com/blog/2016/02/09/static-julia.html, 2016. – [Online; accessed 23-May-2018]

[Dan17] DANISCH, Simon: *Funktionsorientiert und schnell: Die Programmiersprache Julia*. https://www.heise.de/developer/artikel/Funktionsorientiert-und-schnell-Die-Programmiersprache-Julia-3793160.html?seite=all, 2017. – [Online; accessed 23-May-2018]

[Far17] FARBER, Rob: *Julia Language Delivers Petascale HPC Performance*. https://www.nextplatform.com/2017/11/28/julia-language-delivers-petascale-hpc-performance/, 2017. – [Online; accessed 22-May-2018]

[hei16] HEISE.DE, Alexander Neumann @.: *Programmiersprache: Julia profiliert sich beim High Performance Computing | heise Developer*. https://www.heise.de/developer/meldung/Programmiersprache-Julia-profiliert-sich-beim-High-Performance-Computing-html, 2016. – [Online; accessed 22-May-2018]

[JC16] JULIA COMPUTING, Inc.: *Julia for Astronomy - Parallel Computing with Julia on NERSC Supercomputer Increases Speed of Image Analysis 225x – Julia Computing*. https://juliacomputing.com/press/2016/11/28/celeste.html, 2016. – [Online; accessed 22-May-2018]

[JC17] JULIA COMPUTING, Inc.: *Parallel Supercomputing for Astronomy –Julia Computing*. https://juliacomputing.com/case-studies/celeste.html, 2017. – [Online; accessed 22-May-2018]

[Lan14] LANG, Julia: *Microbenchmarks*. https://github.com/JuliaLang/Microbenchmarks, 2014. – [Online; accessed 30-June-2018]

[Lan16] LANG, Julia: *An introduction to ParallelAccelerator.jl*. https://julialang.org/blog/2016/03/parallelaccelerator, 2016. – [Online; accessed 28-May-2018]

[Lan17] LANGUAGE, The J.: *JuliaCon 2017 | AoT or JIT: How Does Julia Work? | Jameson Nash*. https://www.youtube.com/watch?v=7KGZ_9D_DbI&t=978s, 2017. – [Online; accessed 23-May-2018]

[Lan18] LANGUAGE, The J.: *The Julia Language*. https://docs.julialang.org/en/stable/manual/, 2018. – [Online; accessed 22-May-2018]

[Pra17] PRABHAT, Regier und F.: *JuliaCon 2017 | Celeste.jl: Petascale Computing in Julia | Prabhat, Regier & Fischer*. https://www.youtube.com/watch?v=uecdcADM3hY, 2017. – [Online; accessed 22-May-2018]

[the14] THENEWPHALLS: *Understanding object-oriented programming in Julia – Objects (part 1)*. https://thenewphalls.wordpress.com/2014/02/19/understanding-object-oriented-programming-in-julia-part-1/, 2014. – [Online; accessed 22-May-2018]

[Wel18] WELCH, Ivo: *Parallel Processing*. http://julia.cookbook.tips/doku.php?id=parallel, 2018. – [Online; accessed 30-June-2018]

[Wik17] WIKIPEDIA: *String interning - Wikipedia*. https://en.wikipedia.org/wiki/String_interning, 2017. – [Online; accessed 23-May-2018]

[Wik18a] WIKIPEDIA: *Abstract syntax tree - Wikipedia*. https://en.wikipedia.org/wiki/Abstract_syntax_tree, 2018. – [Online; accessed 23-May-2018]

[Wik18b] WIKIPEDIA: *Apache-Point-Observatorium – Wikipedia*. https://de.wikipedia.org/wiki/Apache-Point-Observatorium, 2018. – [Online; accessed 22-May-2018]

[Wik18c] WIKIPEDIA: *Julia (Programmiersprache) - Wikipedia*. https://de.wikipedia.org/wiki/Julia_(Programmiersprache), 2018. – [Online; accessed 23-May-2018]

[Wik18d] WIKIPEDIA: *Multiple Dispatch - Wikipedia*. https://en.wikipedia.org/wiki/Multiple_dispatch, 2018. – [Online; accessed 22-May-2018]

[Wik18e] WIKIPEDIA: *Nominal type system - Wikipedia*. https://en.wikipedia.org/wiki/Nominal_type_system, 2018. – [Online; accessed 22-May-2018]

[Wik18f] WIKIPEDIA: *Sloan Digital Sky Survey – Wikipedia*. https://de.wikipedia.org/wiki/Sloan_Digital_Sky_Survey, 2018. – [Online; accessed 22-May-2018]

[Wik18g] WIKIPEDIA: *Type system - Wikipedia*. https://en.wikipedia.org/wiki/Type_system, 2018. – [Online; accessed 22-May-2018]