

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, 1. Juli 2018

---

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
<b>2</b>	<b>Einführung in die Programmiersprache Julia</b>	<b>3</b>
2.1	Übersicht . . . . .	3
2.2	Typensystem und Multimethoden . . . . .	3
2.3	Metapr. - eine Hommage an Lisp . . . . .	5
2.4	JIT Compiler auf Basis von LLVM . . . . .	5
2.5	Interne und externe Bibliotheken . . . . .	6
<b>3</b>	<b>Parallele Programmierung mit Julia</b>	<b>6</b>
3.1	Remote-Aufrufe und -Referenzen . . . . .	7
3.2	@Spawnat und @Spawn . . . . .	7
3.3	Parallele Map und For-Schleife . . . . .	8
3.4	Shared und Distributed Arrays . . . . .	9
3.5	vertiefende Themen zum parallelen Programmieren . . . . .	9
3.6	Softwarewerkzeuge . . . . .	10
<b>4</b>	<b>Bewertung der HPC-Eignung von Julia</b>	<b>10</b>
4.1	Laufzeittests gegen C und Python für sequenzielle Probleme . . . . .	11
4.2	Laufzeittests für parallele Performanz in Julia . . . . .	13
<b>5</b>	<b>Schlusswort</b>	<b>18</b>

Das ist eine Referenz auf Klein [?]. Die Referenzliste ist in der Datei references.bib definiert.

## 1 Einleitung

1998 startete das Apache-Point-Observatorium in New Mexico(USA) ihr Projekt Sloan Digital Sky Survey (SDSS), 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 als produktivste Weltraumkamera der Welt bis zu ihrer Abschaltung im November 2009. Mit etwa 200 GB reine Bilddaten pro Nacht entstand so eine Datenbank mit über 5 Millionen Bildern von jeweils 12 Megabyte - zusammengerechnet also rund 55 Terabyte<sup>1</sup>. 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<sup>2</sup>. 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. Besonders ist an Julia vorallem ihr Anspruch, eine sehr produktive (high-programming language) 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(KNL) Knoten des Cori Supercomputers des NERSC. Dabei erzielten sie mit Julia eine Peak-Performance von 1.54 PetaFlop/s für Berechnungen mit doppelter Genauigkeit. Damit erzielte das Celeste Projekt einen 100-fachen Performanzgewinn 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, die benötigt werden, um solch eine Performanz zu erreichen<sup>3</sup>. Umso erstaunlicher ist an diesem Ergebnis also die Tatsache, dass das Team hinter Celeste ihre Geschwindigkeit ausschließlich unter Nutzung ihrer Kenntnisse, dem Julia-Code und dessen Threading-Model erzielen konnten.

Wie in der Abbildung 1 zur pro Thread Auslastung zu sehen, lieferte Julia 82.3 % der Performance. Damit stellt Julia unter Beweis das es sowohl für TheraByte-große Datensätze, wie auch für PetaFlop/s-schnelle Berechnungen geeignet ist. Keno Fischer, CTO von Julia Computing, sagte 2017, das 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

<sup>1</sup><https://juliacomputing.com/press/2016/11/28/celeste.html>

<sup>2</sup><https://juliacomputing.com/case-studies/celeste.html>

<sup>3</sup><https://www.nextplatform.com/2017/11/28/julia-language-delivers-petascale-hpc-performance/>

Abbildung 1: Eine Abbildungsbeschreibung

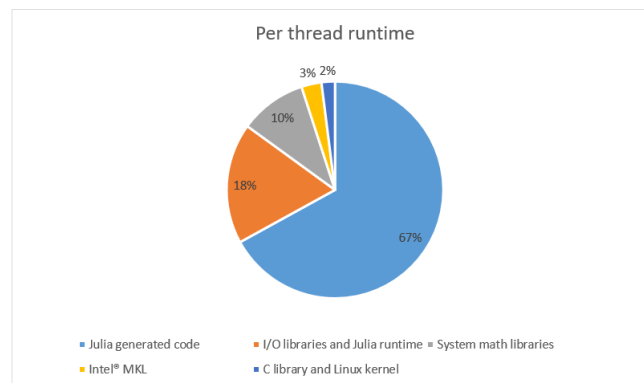


Figure 4: Per-thread runtime performance breakdown

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, dass Julia die Entwicklung in der Forschung für viele Jahre sehr voranbringen wird. Viral Shah, CEO von Julia Computing, sagte das Forscher sich nun auf die Lösung ihrer Probleme fokussieren können und sich nicht mehr mit dem Programmieren beschäftigen müssen<sup>4</sup>.

## 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 Geschwindigkeit von C zu kombinieren. Auch wenn sie vor allem für wissenschaftliches und numerisches Programmieren 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 einem Syntax der 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* erinnert, hat sich Julia viele bereits etablierte Funktionen zunutze 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 vor allem durch Multimethoden für einzelne Funktionen und die automatische Generierung von effizientem typspezialisierten Code gewonnen. Dabei ist Julia sowohl dynamisch wie auch statisch typisiert. Zudem ist Julia funktional, objektorientiert und imperativ und wurde fast komplett in Julia selbst geschrieben.

### 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 reinen 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 An-

<sup>4</sup><https://juliacomputing.com/press/2016/11/28/celeste.html>

sammlung 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 standardmäßig für jeden *Composite Type* auch Konstrukturen. Auch wenn Julia dynamisch ist und damit der Typ nicht angegeben werden muss, besitzt jeder Wert einen festen Datentyp. Dies gilt auch für die vom Nutzer erstellten Typen. Dabei sind alle Datentypen Laufzeittypen, das heißt den einzigen Datentyp den ein Wert besitzt, ist sein wirklicher aktueller Typ, während das Programm läuft. 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*, welcher kein Kindknoten ist. Ein Nutzer kann seine definierten Objekte in diesen Graphen einfügen. Wird nichts weiter spezifiziert, wird das Objekt standardmäßig als Kind vom Typ *Any* angenommen. Als ein weiteres Feature unterstützt Julia auch parameterisierte Datentypen. Diese ermöglichen die generische Erstellung von vielen kombinierten Datentypen 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 wiederum 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 wurde, die der Anzahl der Argumente und deren Typen entspricht. Wird keine gefunden, so wird ein Fehler geworfen. Multimethoden kombiniert mit dem 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 gut auf die Wirklichkeit mehr 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 würde jeder konkrete Typ einem separaten Blattknoten zugeordnet werden und Typen mit ähnlichen Eigenschaften unter einem abstrakten Typ zusammen gefasst werden.

## 2.3 Metapr. - 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 zugeordnet. 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 vor allem 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.

## 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 generieren. 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 deutlich 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 erneut 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.

## 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 verschiedenste 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, dass in der nächsten Version 0.7 Julias Paketverwalter um eine 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.

`verblccall()` und da der von Julias JIT 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.

## 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 sehr viele einfachen Funktionen für simple Probleme, aber gleichzeitig auch eine Tiefe der Funktionen, um das letzte bisschen Performanz noch rauszukitzeln. Als Grundbaustein für Julias Multiprozessorhandhabung dient dabei eine eigene Implementierung einer Nachrichtenübertragung zur Kommunikation zwischen den Prozessen, um diese parallel in separaten Speicherbereichen laufen zu lassen. Diese Übertragung ist anders als bestehende Konzepte wie MPI und ist so ausgerichtet, dass generell nur 1 Seite programmiert werden muss um eine Kommunikation aufzubauen und zu beenden und zusätzlich auch eher wie ein Funktionsaufruf aussieht. 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 1. Solange es mehr Prozessoren als einen 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`, wo bei *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 Arbei-

ter beziehungsweise die Anzahl der Arbeiter zurückgegeben. `rmprocs(id)` entfernt hingegen nur den *Worker* mit der spezifizierten Id.

### 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 gespeichertes 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 dann mittels eines `fetch()` geholt werden, sobald das Ergebnis vorliegt. Ein wichtiger Hinweis ist hier zudem, dass ein einmal mit `fetch()` geholtes *Future* lokal gespeichert wird und deshalb bei einem erneuten `fetch()` keine weitere Kommunikation anfällt. *RemoteChannel* wiederum sind wiederbeschreibbare 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 einen 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 wie der Name es suggeriert ein *Fetch* ausführt.

### 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 mit dem ersten 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, dass 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.

### 3.3 Parallele Map und For-Schleife

Der obere Teil der parallelen 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, in denen er manuell eine Berechnung auslagert. 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 mit 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 häufig Kopf(n) bei M Münzwürfen auftrat, lässt sich beispielsweise sehr einfach mit `@parallel` Parallelisieren und nutzt zudem die Addition als Reduktion

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

Wichtig zu beachten ist hierbei das, 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 globalerweise Variablen sich so nicht beschreibbar 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 noch kein Ergebnis bereit liegt am Ende der Schleife. Hilfreich hierfür ist das Macro `sync`, welches vor das Macro `parallel` geschrieben wird um den Prozess zu blockieren, solange wie alle Berechnungen parallel ausgeführt werden. Als zweites Werkzeug stellt Julia eine parallele Map bereit, welche besonders nützlich für große Funktionen, die einzeln auf eine Menge von Elementen angewandt werden muss, ist. `pmap(i-> log(i), 1:M, batch_size)` berechnet zum Beispiel den Logarithmus der Zahlen eins bis M und speichert sie erneut in einem Array ab. Da `pmap()` vor allem für größere Funktionsaufrufe gestaltet worden ist, kann es hier teils extreme Performanzeinbußen 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 festgelegt werden pro Funktionsaufruf an einen anderen Prozessoren festgelegt werden. Die Nutzung von `@parallel`, das für kleine Iterationen optimiert ist, wäre hier aber genauso möglich.

### 3.4 Shared und Distributed Arrays

*Shared Arrays* nutzen gemeinsam genutzten 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* vor allem 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 Standardfeldern konvertieren bzw den lokalen Anteil des *Distributed Arrays* in ein normales Array extrahieren.

### 3.5 vertiefende Themen zum parallelen Programmieren

Neben diesen doch recht einfachen anwendbaren Funktionalitäten bietet Julia auch noch eine Reihe weitere Möglichkeiten zusätzlichen Performanzgewinn zu erzielen. Die erste Möglichkeit ist dabei die manuelle Handhabung der Lebenszeit der Objekte. Auch wenn der implementierte Müllsammler (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 lokale Instanzen oder auch Remote-Referenzen oder Shared Array aus dem Speicher zu löschen, da der Remote-Knoten sehr groß sein kann und zudem gerade größere Objekte zuerst vom Garbage Collector gelöscht werden. 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 Cluster-managers. 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 standardmäßigen 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 welches mit dem `@acc` Macro ein Mittel bietet, bestimmte Teile wie in diesem Fall ressourcenkostende Arrayfunktionen noch einmal deutlich zu beschleunigen. 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.

### 3.6 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 eigenen Macros `@code_native`, `@code_typed`, `@code_llvm` und vorallem `@code_warntype` welche den herunter compilierten Code für die entsprechende 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 individueller Zeile benötigt worden ist. Auch gibt es einen Linter, welcher Hinweise zum Code gibt und damit aufzeigt, wenn einfachere Implementierungen möglich sind. Ein kurzen Hinweis an dieser Stelle noch bezüglich eines Debuggers, den Julia zwar besitzt, der in seiner derzeitigen Implementierung aber keine Breakpoints oder nativer Code Untersuchung unterstützt und damit bisweilen doch noch etwas ungewohnt zu bedienen ist.

## 4 Bewertung der HPC-Eignung von Julia

Julia erhebt für sich selbst den Anspruch eine sehr performante Sprache zu sein und demzufolge auch Ansprüche als High-Performance- Programmiersprache auf HPCs eingesetzt zu werden und dabei Ressourcen - effizienter oder produktiver in der Produktion als bestehende Sprachen zu sein. Um diesen Anspruch 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 Implementierungen eines sehr gut zu parallelisierenden Problems und eines mittelmäßigem Problems betrachtet. Als Vergleichskriterium diente dabei immer die Laufzeit der einzelnen Funktionen, Zufälligkeiten, Unregelmäßigkeiten und Seiteneffekte wurden versucht so klein wie nur möglich zu halten. Für die sequenziellen Tests wurden die offiziellen Benchmarks der Julia - Lang - Community benutzt und sind als solche auch mit dementsprechender Vorsicht zu betrachten, da Probleme unter Umständen so gewählt worden, das Julia besonders gut aussieht. Nicht desto trotz stellen sie aber ein Vergleichskriterium dar und wurden schlussendlich vorallem gewählt, weil sie nicht die best - mögliche Implementierung in

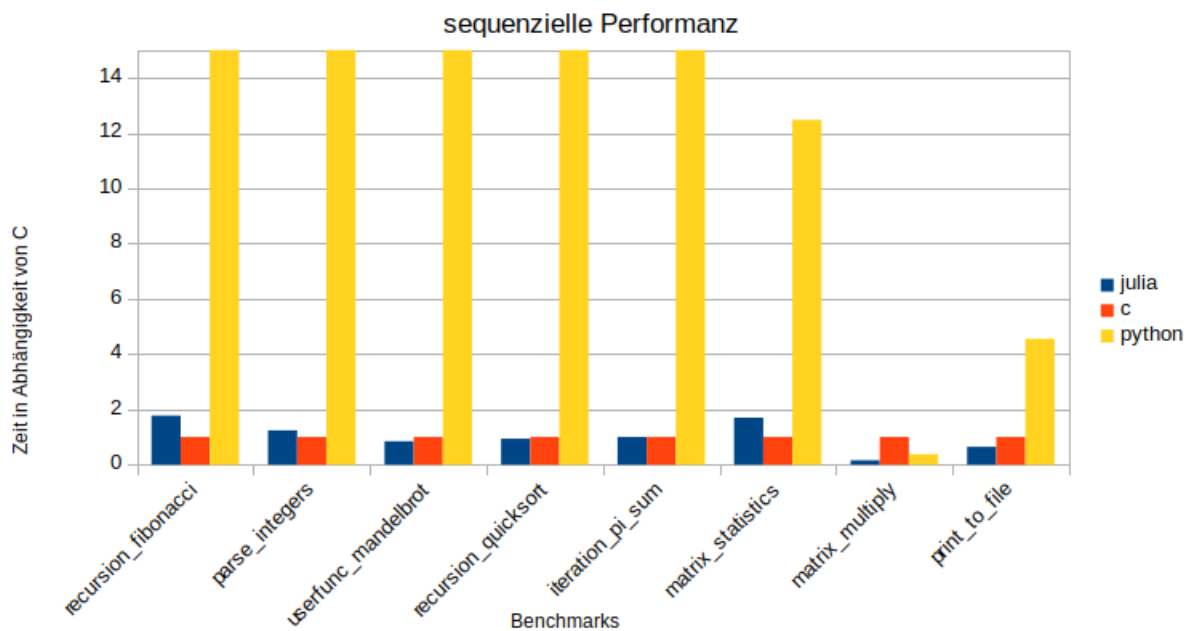
der jeweiligen Sprache darstellen sondern so ähnlich wie nur möglich geschrieben wurden. Alle Ergebnisse für die sequenziellen Laufzeittests und ein Teil der parallelen Tests wurden auf dem Taurus-System der technischen Universität Dresden, explizit einem Haswell-Node mit 24 eingebundenen CPUs und exklusiv Rechten berechnet.

#### **4.1 Laufzeittests gegen C und Python für sequenzielle Probleme**

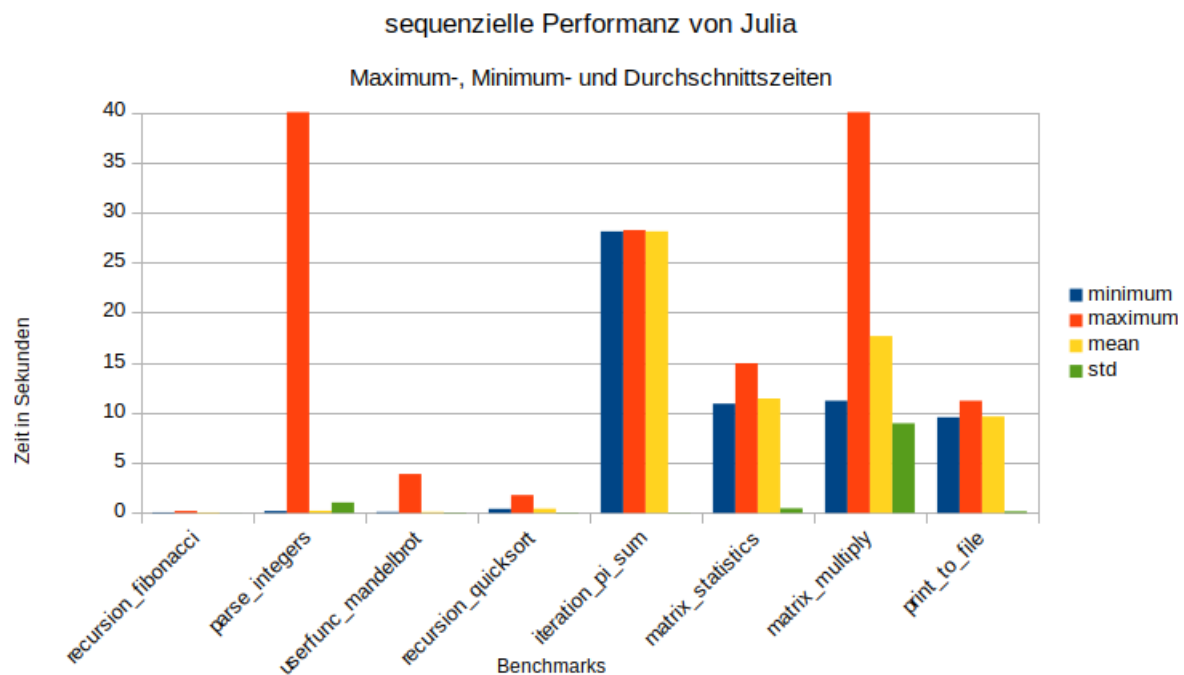
Wie bereits erwähnt wurden für die folgenden Vergleiche von Julia mit anderen Programmiersprachen öffentliche Microbenchmarks von der Julia - Lang - Community benutzt. Dabei wurden lediglich Julia, C und Python miteinander verglichen, da sich Julia dadurch bereits sowohl genügend einordnen lässt als auch diese beiden zusätzlichen Sprachen zudem die Sprachen sind, mit denen sich Julia messen möchte. Julia trat hierbei mit der Version 0.6.2 gegen einen 0.7.1 gcc Compiler und die Python Version 0.3.6 von Anaconda an. Als Benchmarks wurden verschiedene typische Programmmuster getestet wie etwa Funktionsaufrufe, Zeichenkettenzergliederung (engl. String parsing), Sortieren von Daten, numerische 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. Z. B. nutzt der `iteration_pi_sum` Algorithmus 2 für jede Programmiersprache die gleiche For - Schleife und sieht zudem fast komplett identisch zwischen Python und Julia aus.

Abbildung 2: Berechnung der Reihensumme  $1/q^2$  500 - mal in Julia, C und Python von links nach rechts

<pre>function psum()     sum = 0.0     for j = 1:500         sum = 0.0         for k = 1:10000             sum += 1.0/(k*k)         end     end     sum end</pre>	<pre>double psum() {     double sum = 0.0;     for (int j=0; j&lt;500; ++j) {         sum = 0.0;         for (int k=1; k&lt;=10000; ++k) {             sum += 1.0/(k*k);         }     }     return sum; }</pre>	<pre>def psum():     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</pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------



Wie in ?? zu sehen, ist hierbei Python teils deutlich langsamer und verlangte sogar eine Eingrenzung der Skale, da es bei `recursion_fibonacci` z.B. 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, das Julia etwa ähnlich schnell wie C, ja sogar bisweilen schneller als C lief, was aber auch eindeutig den Benchmarks und dessen verminderter Aussagekraft an dieser Stelle zuzuschreiben ist. Allerdings erkennt man z.B. auch an der Stelle für die Matrixmultiplikation, wie schnell Julias Code werden kann, wenn der Compiler in diesem Fall beispielweise wusste, das es sich um das Produkt zweier  $n \times n$  - Matrizen gefüllt mit Einsen handelt. 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, das Julia JIT compiliert und damit auch sehr große Verlustzeiten beim ersten Durchlauf einer Funktion zu erwarten sein können, was gut erkenntlich daran ist, wie unterschiedlich die Maximum - und Minimumlaufzeiten sind??. 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 diesem 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 [?], 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. Neben dem Tarus-System mit Haswell Node wurde das Programm zum Vergleich auch auf einem HP - Omen 17 mit 8 Kernen a Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz und 16GB Arbeitsspeicher lokal ausgeführt um auch einen direkten Vergleich zu Technologien die eher auf sequenzielle Performanz hin optimiert worden sind zu erhalten. Im folgenden Code - Snippet sind die genutzten Verfahren in ihren Funktionen gekapselt zu erkennen. Alle Verfahren wurden benutzt um die Summe über ein Array bzw ein SharedArray 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, wobei gute Parallelisierbarkeit hier wenige Funktionsaufrufe mit schwerer, aber weniger speicherverbrauchende Funktionsberechnungen bezeichnet und mittlere für mehr Funktionsaufrufe mit mittlerer, mehr speicherverbrauchende Funktionen steht. Dies wurde alles über den globalen Parameter `GlobalFunsumTerms` gesetzt, da dieser den Aufwand für die Funktion `fun` festlegte. Gut erkennbar ist zudem, wie ähnlich sich die einzelnen Implementierungen sehen.

Code - Snippets der verschiedenen Verfahren

```

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

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

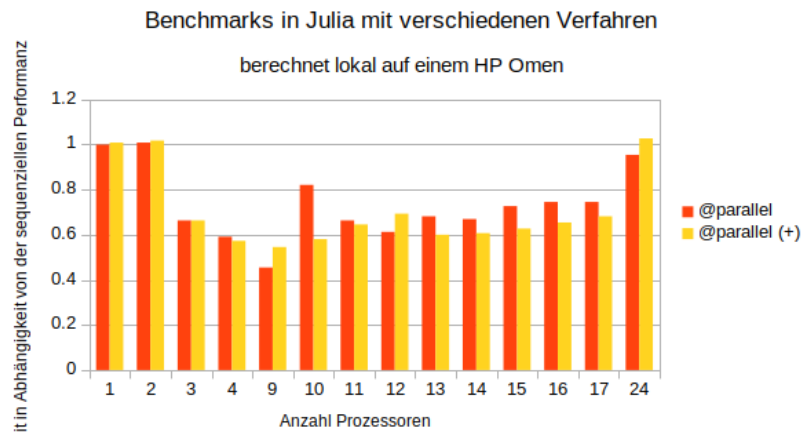
function sharedarray_mapreduce(pt::Int, fun::Function=GlobalFun)
    sa=SharedArray{Float64} (GlobalNumFunCalls)
20
    @parallel (+) for i=1:GlobalNumFunCalls;
        sa[i]= fun(GlobalFunsumterms);
    end#for
    sum(sa)
25 end#function

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

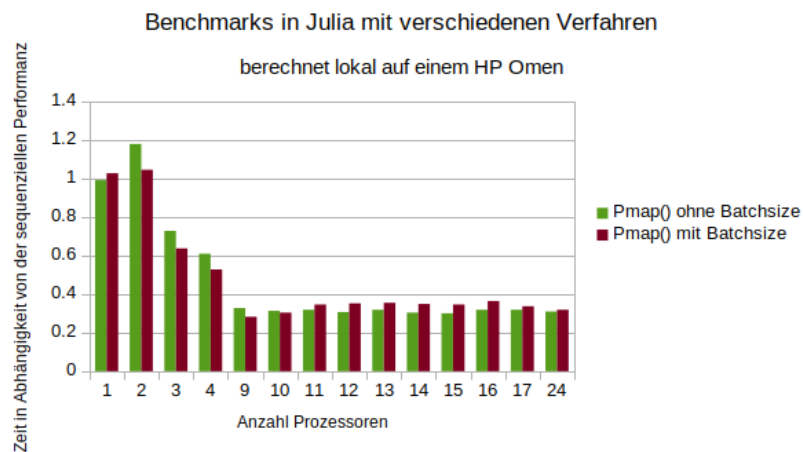
function pmap_sum_b(pt::Int, fun::Function=GlobalFun)
    r = pmap( i-> fun(GlobalFunsumterms), 1:GlobalNumFunCalls,
        batch_size=ceil(Int, GlobalNumFunCalls/nworkers()) )
35    sum(r)
end#function

```

Wie in 3 gut zu erkennen ist, schlägt sich die parallele For-Schleife eher verhalten auf dem HP Omen und erreicht bei 9 Kernen ihr Optimum. Aufgrund der Hardware des Rechners und der Art und Weise, wie der Test geschrieben wurde, wird die parallele For-Schleife hier durch die Anzahl der Kerne begrenzt und hat zudem Probleme, wirklich effizient zu laufen, da das Problem wenige, aber sehr rechenintensive Berechnungen durchführt, in denen Pmap überlegen ist. Der Anstieg beim Hinzufügen von mehr als 9 Kernen lässt sich durch den vergrößerten Kommunikationsaufwand erklären, der allerdings keinen Mehrwert nach sich ziehen kann, da es nur 8 Kerne gibt. Was in dieser Abbildung nicht gut zu erkennen ist, aber sich auf dem Taurus - System andeutungsweise erahnen lässt, ist zudem die effizientere Speicherverwaltung von der Reduktionsfunktion, weswegen diese wenn möglich immer zu preferieren ist. Pmap schlägt sich auf dem lokalen sehr gut und kann einen c. a. 3 - fachen SpeedUp im Optimum bei 9 Kernen erzielen. Auf dem Taurus - System hingegen 4 sind sowohl pmap wie auch @parallel sehr effizient und können einen bis zu 10 - fachen SpeedUp erzielen, wobei pmap dabei nochmal um bis zu Faktor 2 schneller wird. Zudem fällt hier wie schon oben angesprochen auf, dass die Reduktionsfunktion etwas effizienter im Speicher ist und dadurch auch etwas schneller laufen kann. Auch das bei zwei eingebundenen Prozessoren die Performanz schlechter wird als bei einem ist gut ersichtlich und liegt an Julias Implementierung des Master - Worker - Prinzips speziell das der Hauptprozess mit der Id eins zum Master wird und keine Berechnungen mehr durchführt, wohl aber die Kommunikation statt findet zwischen den Prozessoren.



(a) gut zu parallelisierendes Problem mit parallele For - Schleife



(b) gut zu parallelisierendes Problem mit parallele Map

Abbildung 3: Graphen für gute Parallelisierbarkeit lokal berechnet

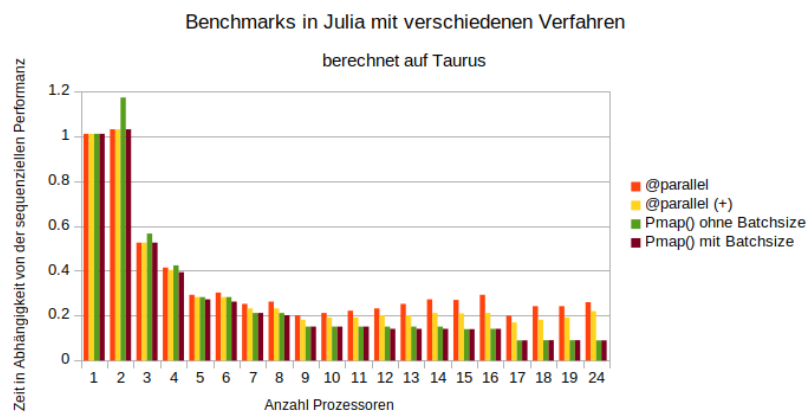
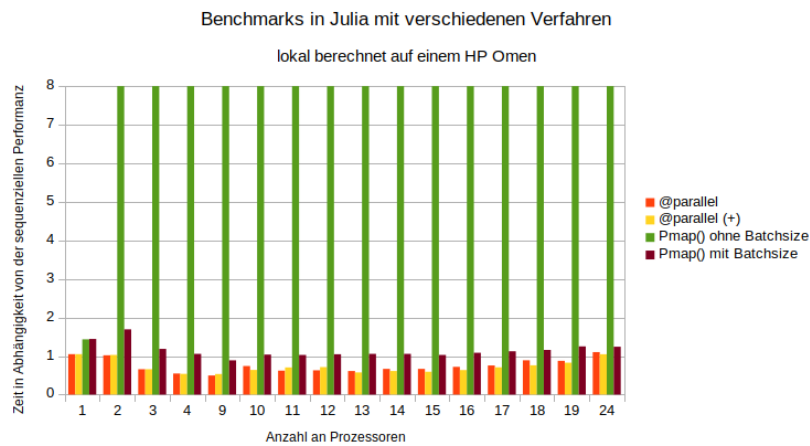


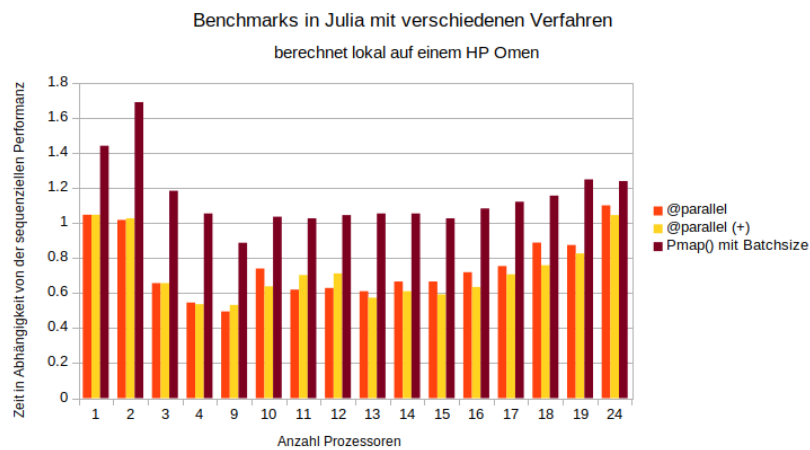
Abbildung 4: Graph für gute Parallelisierbarkeit auf Taurus berechnet



5 Bei mittelmäßig parallelisierbaren Problemen fällt die unterirdische Performanz von `Pmap()` ohne `Batchsize` sofort ins Auge. Mit bis zu 30 - facher Zeit zu sequenziell lokal und zumindest noch Faktor 23 - facher Zeit auf dem Taurus ist die parallele Map ohne die Konfiguration der Batchsize nicht zu gebrauchen und wird deshalb auch in den folgenden Diagrammen entfernt um diese übersichtlicher zu gestalten. Bei der mittleren Parallisierbarkeit schlägt sich die parallele For - Schleife ähnlich zum ersten Durchlauf bei guter Parallelität. Mit einem Faktor von 2 besser als die sequenzielle Performanz als Optimum stellt sie damit vorallem unter Beweis, auch für weniger rechenintensive Funktionen noch geeignet zu sein und diese beschleunigen zu können. Die parallele Map bricht für kleinere Funktionen trotz optimaler Batchsize leider etwas zusammen und erzielt im Optimum gerade so eine bessere Performanz als die sequenzielle Verarbeitung und ist damit für diese Art von Problemen kaum noch geeignet, zumindest für lokale Rechner Zuhause. Auch auf dem Tarus - System 6 verliert die parallele Map deutlich an Geschwindigkeit gegenüber der parallele For - Schleife erreicht aber zumindest noch einen Speedup von 2. Die parallele For - Schleife hingegen wird teilweise sogar noch schneller als für gute Parallisierbarkeit, was nochmal deutlich zeigt, dass dessen Stärke in kleinen Funktionen mit sehr hohen Iterationszahlen liegt und die gute Parallisierbarkeit einfach nicht das Verfahren optimal ausnutzen konnte. Auch kann man im Graphen gut erkennen, dass das mittlere Problem seine Problemgröße so verändert hat, dass bei 15 Kernen das Optimum erreicht ist und danach keine weitere Möglichkeit mehr besteht, den Algorithmus weiter parallel zu beschleunigen, da der Kommunikationsaufwand den Geschwindigkeitsgewinn von weiteren Kernen überwiegt.



(a) mittelmäßig zu parallelisierendes Problem



(b) mittelmäßig zu parallelisierendes Problem ohne 'ohne BatchSize'

Abbildung 5: Graphen für mittlere Parallelisierbarkeit lokal berechnet

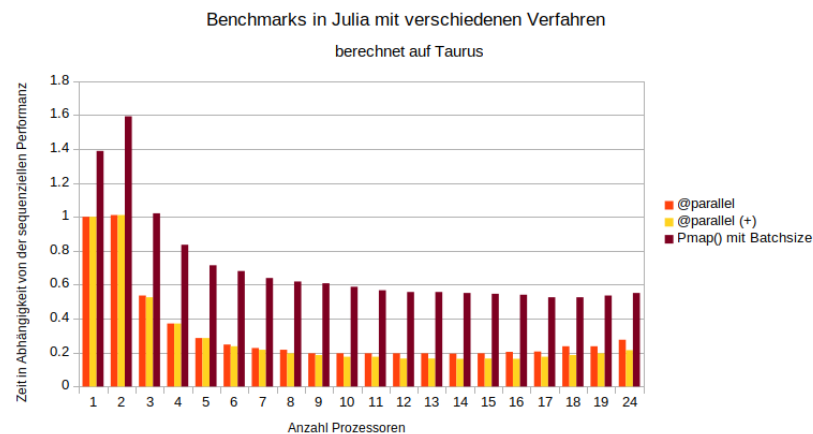


Abbildung 6: Graph für mittlere Parallelisierbarkeit nur mit Batchsized pmap auf Taurus berechnet

## 5 Schlusswort

Festzuhalten bleibt damit im Nachhinein, dass Julia eine sehr praktische und auch schnelle Sprache sein kann. Viele Algorithmen konnte Julia teils deutlich beschleunigen ohne dabei den angenehmen Syntax von Python aufgeben zu müssen. Gerade sequenziell ist Julia teils beeindruckend schnell allein dadurch, dass es dem Compiler mitteilen kann, welche bestimmten Typen ein Wert hat und kann dadurch und durch die Speicherung des Maschinencodes C Performanz fast erreichen, sofern der Nutzer auch weiß, was er tut und Julia so einsetzt, dass es optimal laufen kann. Auch für parallele Aufgaben auf dem HPC scheint es in erster Linie geeignet zu sein und stellt zumindest unter Beweis mit sehr einfachen Syntax weiteren Performanzgewinn zuzulassen und kann damit zurecht behaupten, Python als rapid - Prototyping - Sprache zu verdrängen und durch leichte Optimierungen auf die Auslagerung in C Routinen verzichten zu können. Bestehende Bibliotheken aus anderen Sprachen lassen sich zudem sehr einfach einbinden und ausführen und erreichen dabei zumeist ähnliche Geschwindigkeiten. Auch die Community ist sehr aktiv und hilfsbereit, wodurch sich der Einstieg in diese noch sehr junge Sprache deutlich vereinfachen lässt und man auch schnell an interessante Code - Snippets heran kommt. Ein sehr großer Nachteil den Julia allerdings immer noch hat und der dafür sorgt, dass 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, erforderten einige Anpassungen im Quellcode um den Nightly - build wieder kompatibel mit der Version 0.6.2 zu machen und ähnlich wird es sich auch sicherlich nach Upgrade auf Version 0.7.0 verhalten, welche derzeit in der Alpha ist und kurz vor Veröffentlichung steht. Mit dazu in Zusammenhang steht natürlich 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 voll 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 um alles zum Laufen zu bringen. Dadurch fühlt sich Julia bisweilen auch noch etwas mühselig an. Alles in allem würde ich aber trotzdem dazu raten, einzelne Projekte am HPC mit Julia auszutesten oder zumindest bis zum ersten öffentlichen Release (der nicht mehr lange auf sich warten lassen soll!) zu warten und Julia derweil im Auge zu behalten, da sie wirklich schnell sein kann, wie meine Beispiel aus der Einleitung auch unter Beweis stellt.

## **Erklärungen zum Urheberrecht**

Hier soll jeder Autor die von ihm eingeholten Zustimmungen der Copyright-Besitzer angeben bzw. die in Web Press Rooms angegebenen generellen Konditionen seiner Text- und Bildübernahmen zitieren.