Supersonic Algorithms

Anton Rodenwald (18)

14. Januar 2023

Projektbetreuerin:	Birgit Ziegenmeyer
Institution:	Schillerschule Hannover
Thema des Projektes:	Implementationen von Algorithmen in verschiedenen Programmiersprachen und Analyse dieser in Bezug auf die besten Techniken zur Optimierung, um die Umsetzung hochperformanter und effizienter Programme zu erforschen.
Fachgebiet:	Mathematik/Informatik
Wettbewerbssparte:	Jugend Forscht
Bundesland:	Niedersachsen
Wettbewerbsjahr:	2023

Kurzfassung

Durch den Informatikunterricht bekam ich die Idee, zu erforschen, wie sich die Implementation von Algorithmen optimieren lässt. Relevanz hat die Optimierung der Implementation, weil sich durch effiziente Programme Zeit, Geld und z. B. in Rechenzentren auch Strom sparen lässt. Einen besonderen Fokus legte ich dabei auf die Sprache Python im Vergleich mit C/C++. Meine Erwartung war, das Python deutlich langsamer sein würde als C/C++, was auch der Meinung des Internets entsprach. Ich nutzte die den Sortieralgorithmus Quicksort als Beispiel und implementierte viele optimierte Versionen dieser in Python unter Nutzung der Bibliotheken NumPy, Numba, Cython und CTypes. Nach Messung der Ausführungszeit konnte ich feststellen, dass Python mit NumPy und Numba so schnell wie C++ werden kann. Außerdem fiel mir auf, wie stark das Zusammenspiel zwischen Python und C/C++ ist und wie sich interpretierte und kompilierte Programmiersprachen unterscheiden.

Inhaltsverzeichnis

1 Einleitung

In meinem Projekt wollte ich herausfinden, wie sich Programme durch geschickte Implementation in ihrer Ausführung beschleunigen lassen. Ich fragte mich, welche Optimierungen den größten Zuwachs an Performance bringen. Diese Frage schien mir relevant, da durch effiziente Programme Zeit, Geld und Strom z. B. in Rechenzentren gespart werden können. Es ging ausdrücklich nicht darum, verschiedene Algorithmen auf theoretischer Ebene miteinander zu vergleichen, da dies bereits zu Genüge gemacht wurde. [**sortieralgorithmenwikipedia**] Die Idee entstand dadurch, dass ich im Herbst 2022 im Informatikunterricht Sortieralgorithmen wie die von Hoare entwickelte Quicksort kennenlernte. [**quicksortwikipedia**] Ich stellte mir die Frage, wie man eine Liste von 10 Millionen zufällig generierten positiven Integern am schnellsten sortieren kann. Da algorithmisch bei der Quicksort keine Verbesserung möglich schien, kam ich so auf das Thema, die Ausführung möglichst zu beschleunigen, indem ich mich mit Implementationsoptimierungen beschäftigte. Dabei wollte ich auch ergründen, wodurch sich Unterschiede in der Performance zwischen Programmiersprachen erklären lassen. Meine Hypothese war dabei, dass C/C++ deutlich schneller als andere Sprachen wie z. B. Python sein würde und dass ich in Python die Performance nicht so stark optimieren kann. Als Programmierer bemühte ich nun das Internet und fand 4 Quellen, die C/C++ als 10x bis 100x mal schneller beschrieben, was sich mit meiner Vermutung deckte [**pyengineeringvscpp**] [**quorepythonvscpp**] [**stopythonvscpp**]. Auch fand ich einige Artikel von eifrigen Programmierern, die bereits Dinge zur Optimierung unternommen hatten. Ich wurde so zur Nutzung von Cython, CTypes, Numba und NumPy bewegt, wovon die letzteren beiden mit bereits bekannt waren [**cythonctypes**]. Meine Internetsuche nach den besten Optimierungen für Python ergab nur Tipps, aber keine guten Vergleiche der Geschwindigkeit, weshalb ich entschied, dass ein Vergleich der Optimierungsmöglichkeiten sinnvoll sein könnte. [**pythonopt1**] [**pythonopt2**] [**pythonopt3**].

2 Materialien, Vorgehen, Methode

2.1 Materialien

Für mein Projekt nutze ich meinen Desktop PC (Linux) zum Implementatieren und Ausführen der Programme. In diesem verbaut sind eine Ryzen 7 2700 und 16 GB Ram. Außerdem nutzte ich mehrere Programme. Bis auf Visual Studio Code (Implementierung

der Programme) und LaTeX(Erstellen der schriftlichen Arbeit) waren dies die Compiler oder Interpreter zum Ausführen der verschiedenen Programmiersprachen. Diese Programme waren (mit Versionsangabe) Clang++ 14.0.6 (C/C++ Compiler), Python 3.10.8 (Ausführen von Python Programmen), Java 19.0.1 (Kompilieren und Ausführen von Java), Lua 5.4.4 (Ausführen von Lua Programmen), Nodejs 18.8.0 (Ausführen von Javascript Programmen), Julia 1.8.3 (Ausführen von Julia Programmen) und Go 1.19.4 (Go Compiler).

2.2 Vorgehen und Methode

Mein Vorgehen war für alle getesteten Programmiersprachen ähnlich. Zuerst generierte ich in der jeweiligen Sprache eine unsortierte Liste bzw. Array mit 10 Millionen Pseudo-Zufallszahlen, die anschließend von einer Quicksort Implementation in dieser Sprache aufsteigend sortiert wurden. Da jede Sprache über Möglichkeiten der Zeitnahme verfügt, konnte ich die zur Sortierung benötigte Zeit ermitteln und mir auf der Konsole ausgeben lassen. In Python implementierte ich dazu eine Timer-Klasse (Abb. 1). Diese konnte ich im Quellcode nutzen, um die Zeit von Programmcodeabschnitten zu messen (Abb .2). Die Ergebnisse dieser Messungen ließ ich mir auf der Konsole ausgeben (Abb. 3).

```python
import time

class Timer:
    timers = {}
    @staticmethod
    def start(name):
        Timer.timers[name] = time.time_ns()

    @staticmethod
    def stop(name):
        diff = time.time_ns() - Timer.timers[name]
        diffseconds = diff / (10 ** 9)
        print(name, ":", round(diffseconds, 5), "Seconds")
```

Abb. 1 Timer Klasse

```python
Timer.start("standard list.sort() on numpy array")
liste.sort()
Timer.stop("standard list.sort() on numpy array")
```

Abb. 2 Zeitnahme

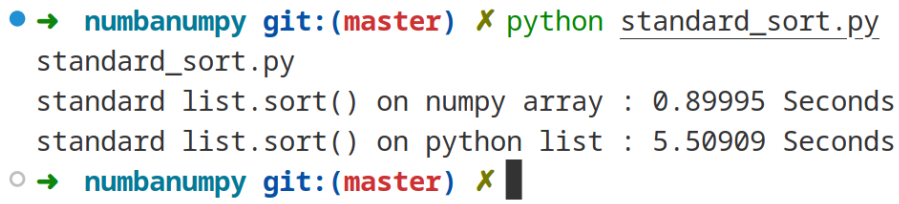


Abb. 3 Beispielausgabe

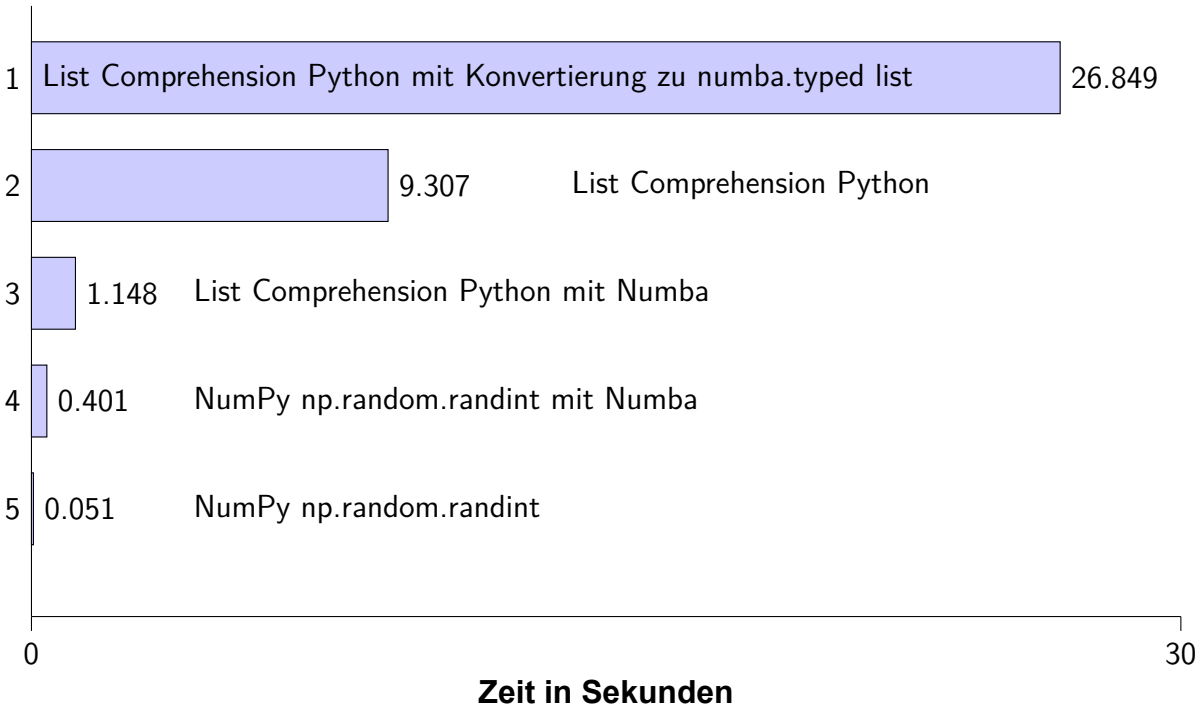
In den Sprachen Lua, Java, Javascript, Julia, Go und C/C++ nahm ich dabei keine weiteren Optimierungen vor, sondern fokussierte mich voll und ganz auf die Optimierung von Python. Ich nutzte dabei die Bibliotheken NumPy, Numba, CTypes und Cython, über die ich mich, nach Programmierer Manier, durch das Nutzen einer nicht auflistbaren Anzahl an Online-Ressourcen und durch das Lesen der Dokumentation informierte [**cythondocs**] [**cythondocsnumpy**] [**cythonctypes**]. Eine Schwierigkeit dabei war, dass ich kaum Erfahrung mit diesen Python-Bibliotheken hatte. In Python entwickelte ich dann eine Vielzahl an verschiedenen Funktionen zur Generierung von Zufallszahlen und implementierte auch viele verschiedene Versionen der Quicksort. Dabei kombinierte ich geschickt die genannten Bibliotheken und Python Standard-Funktionen, um neue Variationen zu schaffen, die potenziell bessere Performance bieten. Ich probierte dabei einfach herum nach dem Prinzip Trial and Erroründ schaute, was die Performance erhöhte. Meine Versionen waren somit Weiterführungen der Ansätze aus diesen Bibliotheken, aus denen ich verschiedene Optimierungsmöglichkeiten nutzte. Ein Beispiel sind die Bibliotheken Numba und NumPy. Beide brachten meinen Versionen einen Geschwindigkeitsboost, doch durch die Kombination beider wurde die Zeitersparnis noch größer. Jede dieser Versionen führte ich nun mehrmals aus und nahm eine der Zeiten, die der ungefähre Mittelwert dieser zu sein schien. So hatte ich nun für jede Version eine Ausführungsdauer. Diese ist zwar keine arithmetischer Mittelwert, aber ausreichend genug zum Vergleichen der Implementationen. Wichtig dabei war, alle Tests auf dem gleichen Computer zu machen, da sich bei unterschiedlicher Hardware die Ergebnisse unterscheiden. Diese wären sonst nicht vergleichbar gewesen. Am Ende des Projektes implementierte ich außerdem noch eine Radixsort in C++ mithilfe einiger Internetquellen [**terdiman**] [**michael**] [**intelavxdocs**] [**avxguide**]. Dies tat ich, um herauszufinden, was die allerschnellste Möglichkeit zur Sortierung von 10 Millionen Zahlen ist. Dabei muss gesagt sein, dass die Radixsort ein anderer Algorithmus ist und somit nicht mit den anderen Programmen mit Quicksort vergleichbar ist.

2.3 Schwierigkeiten

Die größte Schwierigkeit war, dass ich auf diesem Themengebiet noch nicht sehr viel Vorerfahrung hatte. Außerdem waren die Konzepte teils komplex und es war nicht immer einfach, im Internet gute Beispiele zu finden. Vor dem Projekt hatte ich mich z. B. noch nie mit der Python Bibliothek CTypes beschäftigt und deshalb fand ich meist erst nach längerem Suchen im Internet eine Lösung für auftretende Fehler. Auch bei der Implementation von Programmen in C++ hatte ich teils meine Schwierigkeiten, da das Verstehen von einigen Bitoperationen erstmal ein Eindenken in die Thematik erforderte. Ein für mich nicht lösbares Problem war auch die Implementierung in Go. Dort war das Sortieren von mehr als 7 Millionen Zahlen mit einer rekursiven Quicksort leider nicht möglich. Ich fand heraus, dass dies an der Implementierung von Rekursion in Go liegt, die viel Arbeitsspeicher verbraucht, was ein Problem war, da der Arbeitsspeicher für eine Funktion auf 1 Gigabyte begrenzt ist. Diese Grenze wurde ungünstigerweise dann erreicht, wodurch das Programm abstürzte. [**godeeprecursions**] [**goroutinesize**].

3 Ergebnisse

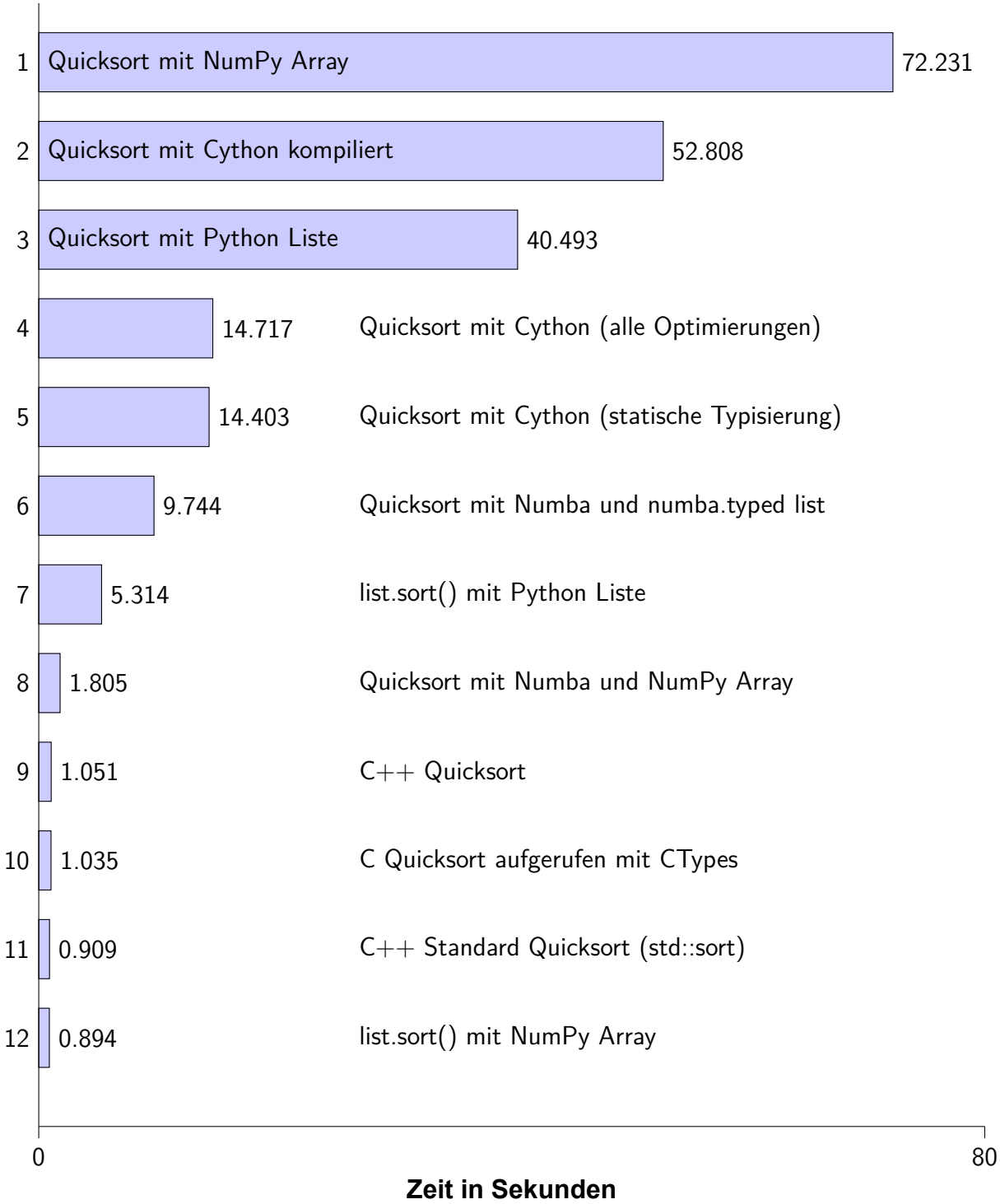
3.1 Generierung von 10 Millionen zufälligen Zahlen in Python



Wie im Vorgehen beschrieben generierte ich unsortierte Listen von Zufallszahlen, damit ich testen konnte, wie schnell meine Quicksort Implementationen diese sortierten.

Die Version mit List Comprehensions in Python (2) dauerte relativ lange. Dies war ein Hindernis beim Entwickeln und Testen der eigentlichen Implementationen der Quicksort. Deshalb optimierte ich die Generierung mit Numba (3, 4) und NumPy (4, 5). Es zeigte sich, dass alle 3 optimierten Versionen deutlich schneller waren. Dies entsprach meinen Erwartungen, dass die Bibliotheken mehr Performance bringen. Auffallend war dabei, dass die Kombination von NumPy und Numba langsamer war, als NumPy allein. Am längsten dauerte die Generierung mit List Comprehensions und anschließender Konvertierung zu einer typisierten Numba Liste (5). Diese Version entwickelte ich allerdings nur, weil meine Optimierung der Quicksort nicht mit normalen Python-Listen kompatibel ist. Für Numba müssen die Listen nämlich in einem kompatiblen Format vorliegen, weshalb die Konvertierung nötig war.

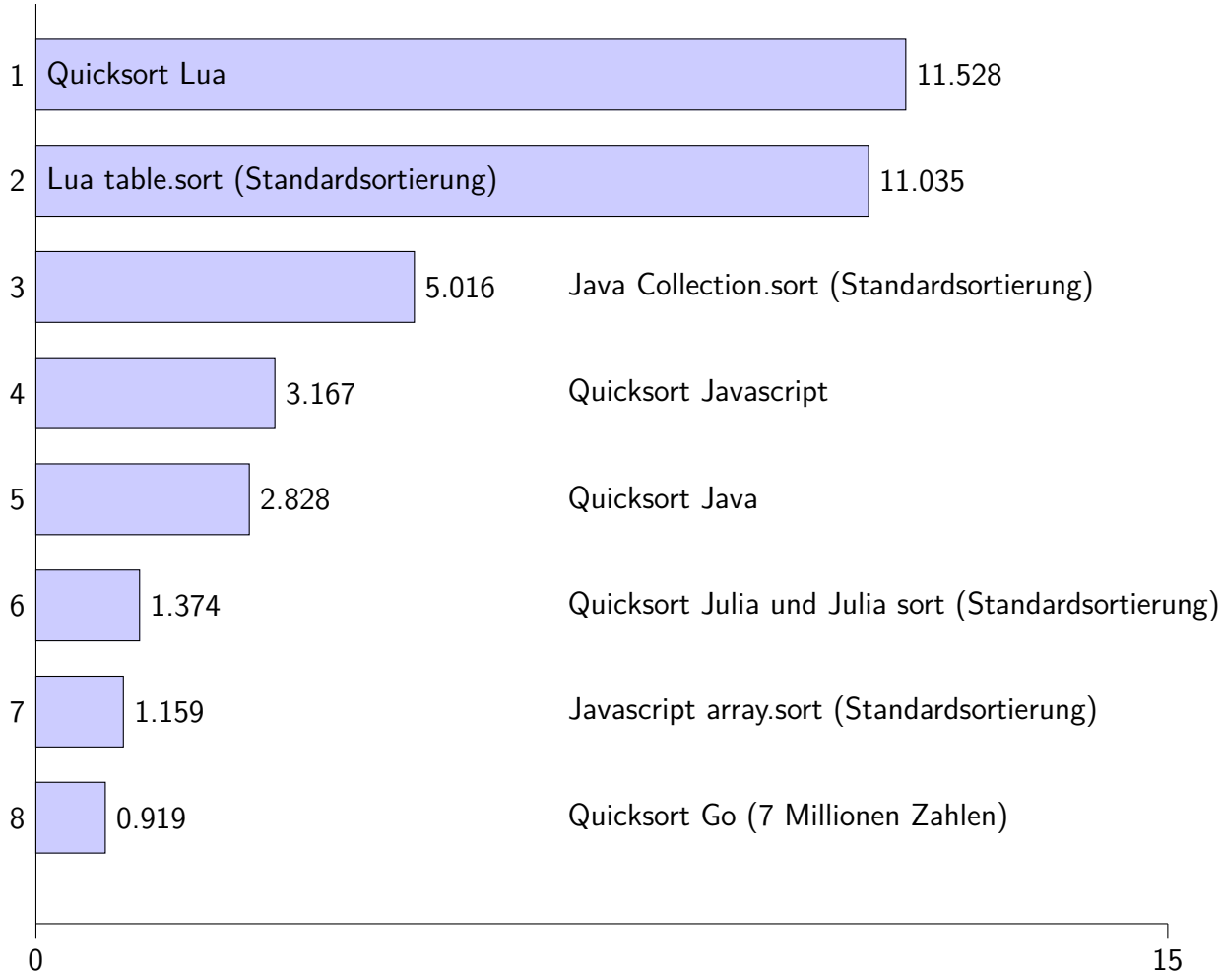
3.2 Quicksort Implementationen in Python mit verschiedenen Optimierungen und Vergleich mit C++



In diesem Diagramm sind meine verschiedenen Versionen der Quicksort in Python, die standardmäßig verfügbaren Sortierfunktionen von Python und C++ sowie eine in C++ implementierte Quicksort zu sehen. Alle diese Versionen ließ ich eine vorher generierte, unsortierte Liste aus 10 Millionen zufälligen Zahlen sortieren. Meine im Informatikunterricht entwickelte Version (3) brauchte ca. 41 Sekunden zur Sortierung. Überraschend

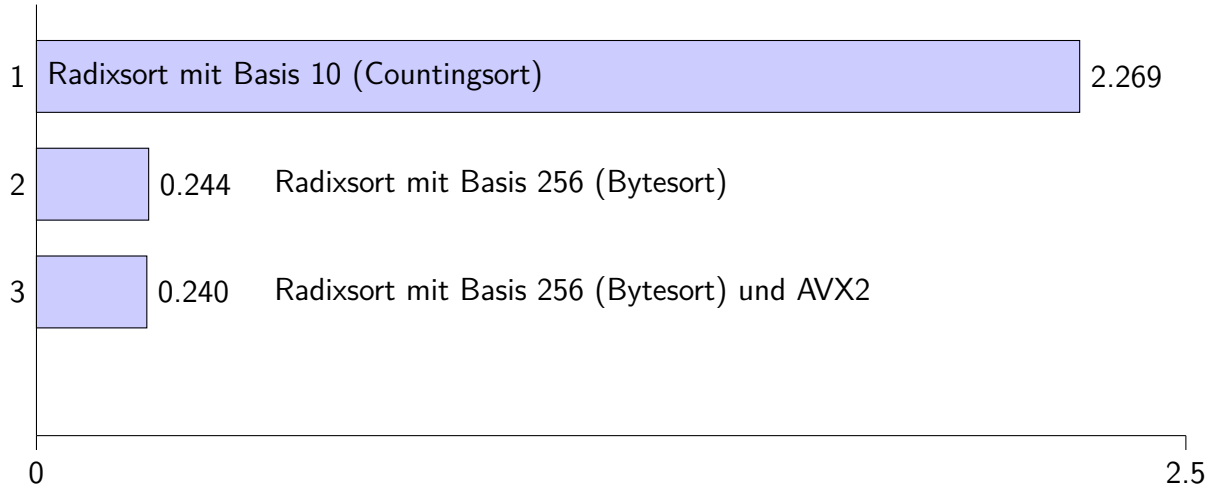
war für mich dann, dass Versionen mit Cython und NumPy (1, 2) teils länger brauchten. Dies hatte ich nicht erwartet. Ich ging eigentlich davon aus, dass sie sich nur positiv verändern würden, also meine Implementationen nur schneller werden würden. Als ich dann Cython zur statischen Typisierung nutzte, zeigte sich ein guter Zuwachs an Performance (5). Weitere Optimierungen mit Cython (4) schienen aber keinen Effekt zu haben, was mich wunderte. Da Cython vor der Ausführung kompiliert wird, hatte ich mit einer größeren Geschwindigkeitserhöhung gerechnet, was nicht der Fall war. Ungefähr 4.5x schneller war meine Version mit Numba und der Numba Typed List (6). Nutze ich Numba nun in Kombination mit einem NumPy Array (8), so erhöhte sich die Geschwindigkeit nochmal um ein Vielfaches. Meine Version mit Numba und NumPy (8) war sogar schneller als die normale Sortierfunktion für Python Listen (7). Dies bestätigte meine Erwartungen, dass NumPy und Numba einen großen Boost in Performance bringen, doch es überraschte mich, dass dieser so groß war. Einzig schneller waren nur einige C/C++ Versionen (9, 10, 11) und die standardmäßige Sortierfunktion für NumPy Arrays, was ich so erwartet hatte. Meine in C++ implementierte Quicksort (9) war dabei ähnlich schnell wie die standardmäßige C++ Sortierfunktion (11). Wichtig zu erwähnen zur Version mit CTypes (10) ist, dass die Konvertierung einer Python Liste in ein für C verständliches Format auch nochmal ca. 2 Sekunden Zeit kostete. Es zeigt sich, dass NumPy eine genauso schnelle Sortierung wie C++ ermöglicht. Dies überraschte mich sehr, weil es meiner Hypothese und der allgemeinen Meinung, die ich so warnahm, komplett widersprach. Ich fragte mich wie sich die Versionen 3 und 12 unterschieden. Was sorgt für diesen großen Unterschied an Performance?

3.3 Quicksort Implementationen in weiteren Sprachen im Vergleich



Neben C++ und Python interessierte mich auch, wie schnell das Sortieren mit einer Implementation der Quicksort in anderen Sprachen möglich ist. Ich nahm dabei aufgrund meines limitierten Wissens in diesen Sprachen keine Optimierungen vor. Dies tat ich auch, weil mich interessierte, wie schnell die einfachste Implementation der Quicksort in diesen ist. Es fällt direkt auf, dass Lua (1, 2) sowohl mit der Quicksort als auch mit der standardmäßigen Sortierung im Vergleich sehr schlecht abschneidet. Darauf folgt die standardmäßige Sortierung in Java (3). Diese ist überraschenderweise langsamer als meine Quicksort Implementation in Java (5). Zwischen diesen beiden liegt meine Quicksort in Javascript. Darauf folgt dann Julia (6), wo meine eigene Quicksort fast genauso schnell war wie die standardmäßige Sortierfunktion. Einzig schneller sind nur die standardmäßige Sortierfunktion in Javascript (7) und meine Quicksort Implementation in Go (8). Diese ist allerdings nur bedingt vergleichbar, da ich aufgrund von beschriebenen Schwierigkeiten nur 7 Millionen Zahlen sortieren konnte. Meine Go-Variante lässt sich also nicht als schnellste Version bezeichnen.

3.4 C++ Radixsort Implementationen



In den vorigen Teilen ging es um die Optimierung der Quicksort. Dieser Abschnitt behandelt die Radixsort, ist also nicht mit den Ergebnissen und Überlegen von davor vergleichbar. Es wurden hier nicht nur Optimierungen, sondern auch andere Algorithmen genutzt. Der Grund, warum ich mich auch mit der Radixsort beschäftigte ist, dass ich den schnellsten Weg zur Sortierung von 10 Millionen Zufallszahlen finden wollte. Die Quicksort hat eine durchschnittliche Komplexität von $O(n \log n)$, die Radixsort von $O(n \cdot w)$. Die Radixsort ist also algorithmisch gesehen schneller. Meine erste Implementation der Radixsort (1) brauchte 2.269 Sekunden. Sie war damit also langsamer als viele der Versionen der Quicksort aus den vorigen Diagrammen. Meine weiteren Implementationen (2, 3) mithilfe der Artikel von Pierre Terdimann und Michael Herf waren dann allerdings deutlich schneller als alle Vorigen der Quicksort Versionen [**terdiman**] [**michael**]. Dies zeigte mir, dass die korrekte Implementation von Algorithmen eine Rolle spielt, da meine erste Version zwar einen überlegenden Algorithmus nutzte, aber trotzdem langsamer war. Auch zeigte sich, dass bei korrekter Implementation die Wahl des Algorithmus sehr wichtig ist. Bei meinen besten Versionen der Radixsort und der Quicksort war erstere deutlich schneller. Daraus schloss ich, dass vor der Optimierung der Implementation am besten erst der beste bekannte Algorithmus gewählt werden sollte. Ich fragte mich, wieso die Radixsort nicht häufiger benutzt wird, was ich mir leider nicht beantworten konnte.

4 Ergebnisdiskussion

4.1 Erklärung der Ergebnisse

Zwischen den verschiedenen Implementationen gab es große Unterschiede in Bezug auf die Ausführungsgeschwindigkeit. Wie diese zustande kamen, will ich im Folgenden erklären. Dazu ist es wichtig, dass Konzept von interpretierten und kompilierten Programmiersprachen zu verstehen. Bei einer interpretierten Programmiersprache wie Python wird der Quellcode von einem Programm, dem Interpreter, zur Zeit der Ausführung verarbeitet. Dabei müssen viele Prüfungen durchgeführt werden, damit das Programm richtig ausgeführt werden kann. Bei kompilierten Sprachen hingegen werden alle Prüfungen vor dem Ausführen durchgeführt. Jeder Quellcode in diesen Sprachen muss vor der Ausführung kompiliert werden und in Maschinensprache umgewandelt werden. Im Gegensatz zur Interpretation können dabei auch deutlich mehr automatische Optimierungen vom Compiler vorgenommen werden. Deswegen sind kompilierte Sprachen generell nahezu immer schneller als interpretierte. Die Bibliotheken NumPy, CTypes, Numba und Cython basieren darauf, dass weniger Python-Quellcode interpretiert werden muss. Dies geschieht dadurch, dass diese Möglichkeiten bieten zur Nutzung von kompiliertem Code in Python. Dabei geht jede Bibliothek unterschiedlich vor. NumPy, CTypes und Python-Standardfunktionen bieten Python Zugriff auf schnelle und bereits kompilierte C Funktionen. Numba ermöglicht die Just-in-Time Kompilierung, also bestimmte Teile des Python Quellcodes während der Ausführung für bessere Performance zu kompilieren. Cython ermöglicht diese Kompilierung bereits vor der Ausführung in gewissem Maße. All diese Konzepte bringen Python näher an die kompilierten Sprachen heran, und somit auch an die Geschwindigkeit von C/C++. Ich erfuhr dies durch ein Betrachten des Quellcodes der Bibliotheken und durch Lesen der Dokumentation [**pythonsource**] [**cythondocsnumpy**] [**numbadoc**] [**numpysource**]. Dieses enge Zusammenspiel zwischen Python und C/C++ und die erreichten Geschwindigkeiten von Python überraschten mich und waren nichts, was ich erwartet hatte. Meine Hypothese, dass Python eine Schnecke ist und C/C++ ein Rennwagen, war falsch. Zugleich wurde mir klar, wie stark Python auf C/C++ aufbaut. Ich wusste zwar bereits, dass der Python Interpreter in C geschrieben ist, doch war dies trotzdem eine wirklich neue Erkenntnis für mich.

4.2 Beantwortung der Forschungsfrage

Durch dieses tiefergreifende Verständnis ist mir jetzt klar, dass NumPy, Numba und CTypes die besten Techniken zur Optimierung von Python bieten. Am meisten Performance lässt sich dann erreichen, wenn möglichst wenig Quellcode interpretiert werden muss und viel auf C/C++ Code basiert. Als Beispiel dafür fand ich das Gebiet der künstlichen Intelligenz. Die dominante Sprache dort ist Python, doch die nötige Performance für solch komplexe Anwendungen wird dadurch erreicht, dass alle genutzten Bibliotheken auf C/C++ Quellcode basieren. Zugespitzt lässt sich sagen, dass Python eigentlich nur eine einfache Möglichkeit ist, C/C++ zu nutzen. Um Python schneller zu machen, muss man möglichst aufhören, Python zu nutzen. Genauer meine ich damit, dass man Python nicht nutzen sollte, um komplexe Algorithmen zu implementieren, da wichtige Sprachkonstrukte wie Schleifen in reinem Python einfach zu langsam sind. Ich appelliere deswegen an alle Entwickler, sich bewusst zu werden, wie ihre Programmiersprachen funktionieren, damit Zeit, Geld und Strom gespart werden können.

5 Reflexion und Ausblick

Obwohl sich meine Hypothese nicht bestätigte, konnte ich trotzdem viel lernen und neue Erkenntnisse sammeln. Insgesamt werte ich das Projekt als ein Erfolg. Ich verstehe nun die Funktionsweise von Python deutlich besser und habe auch Erfahrung mit LaTeXsammeln können. Rückblickend denke ich, dass ich bei meinen Ergebnissen bessere Möglichkeiten zum Testen hätte schaffen sollen. Ich hätte genauere Ergebnisse erhalten, wenn ich viele Testergebnisse zu einem Durchschnittswert kombiniert hätte. Außerdem nehme ich mir vor, beim nächsten Mal etwas strukturierte Vorzugehen in Bezug darauf, was ich wie und wieso teste. Als Weiterführung für mein Projekt wäre denkbar herauszufinden, wieso kaum jemand die scheinbar schnellere Radixsort nutzt. Auch könnte man selbst einen Interpreter oder Compiler entwickeln, um noch genauer zu erforschen, wie sich diese unterscheiden.

6 Schlusswort

Meine Ergebnisse lassen sich mehr oder weniger in einem Bild zusammenfassen. Schnelle Implementationen nutzen kompilierte Sprachen wie C/C++, Rust, …


ALWAYS HAS BEEN
WAIT IT'S ALL C
FAST AND EFFICIENT
CODE IN ANY LANGUAGE
imgflip.com