Pro-Seminar

Softwareentwicklung in der Wissenschaft

Python in High Performance Computing

Malte Eickhoff

Abgabe bis 31.08.2020

Betreuer: Hermann Lenhart

Gliederung:

1. Was ist Python [ab S. 3]

2. Was ist High Performance Computing [ab S. 5]

3. Wie wird Python in HPC realisiert [ab S. 7]

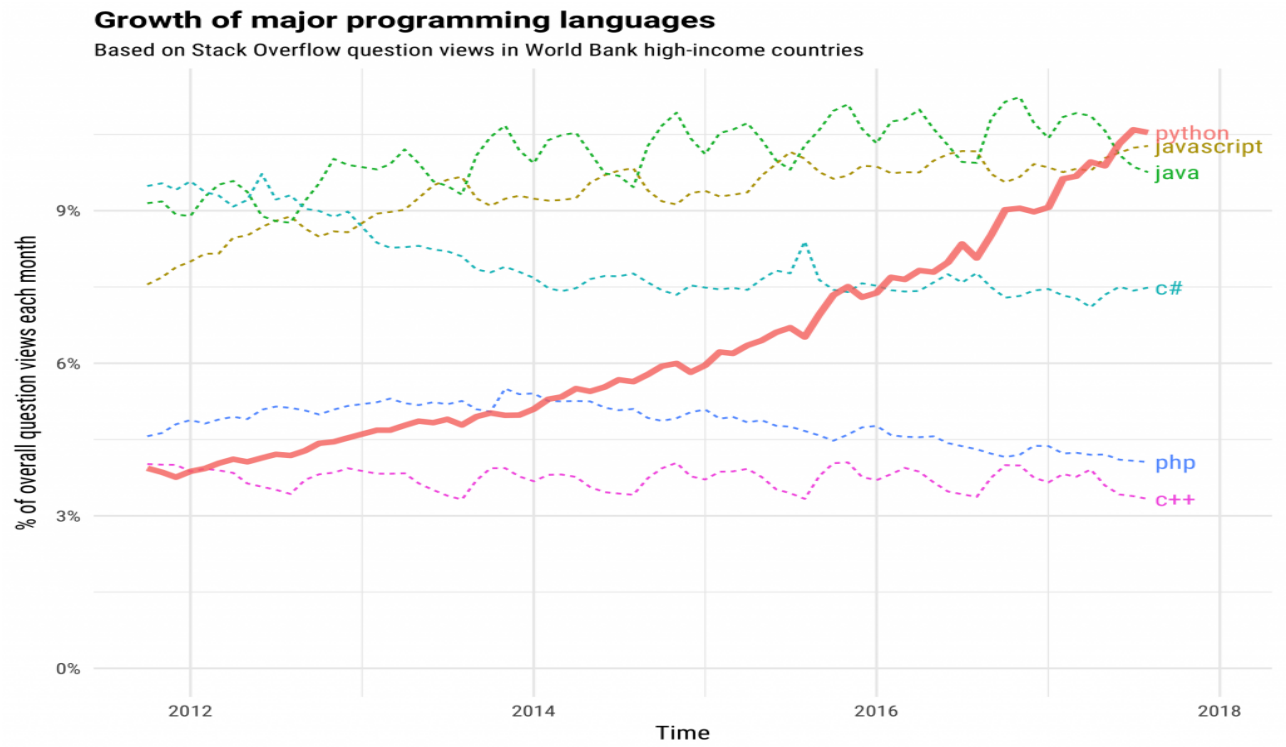
4. Warum Python in HPC [ab S. 14]

5. Fazit [ab S. 16]

6. Quellen [ab S. 17]

1. Was ist Python? Einführung

Python ist eine hochgradig dynamisch typisierte universelle Programmiersprache. Python genießt große Popularität in den Bereichen der künstlichen Intelligenz und Deep Learning. Python als Programmiersprache sah, wie man in Abbildung 1 sehen kann, in den letzten 10 Jahren generell einen stetigen Anstieg an Nutzern weltweit.



[Abb. 1] https://stackoverflow.blog/2017/09/06/incredible-growth-python/

Python als Programmiersprache besitzt einige Vorteile. Gerade für Anfänger und Einsteiger in der Programmierung ist Python von Vorteil, da die Syntax sehr leicht lesbar und intuitiv gestaltet ist. Allgemein ist Python aufgrund der Syntax eine verhältnismäßig ordentlich aufgebaute Sprache. Weiterhin besitzt Python dynamisch typisierte Variablen, was den Nutzern die Anwendung von Variablen deutlich erleichtert und mögliche Fehlerquellen bei der Deklaration verhindert.

Ebenso nutzt Python eine automatische Speicherverwaltung, was ebenfalls die Anwendung erleichtert.
Jedoch hat Python auch Nachteile, wie zum Beispiel die langsamen Kompilierungszeiten. Nach dem “leichten” Einstieg in Python kann die tiefere Nutzung ebenfalls schwer fallen, aufgrund der dezentralen Organisation von Python, mittels Paketen etc., ist die Übersichtlichkeit über die Menge and Modulen und Möglichkeiten mit dieser Programiersprache für Neueinsteiger nicht ganz so gut gegeben.

Syntax

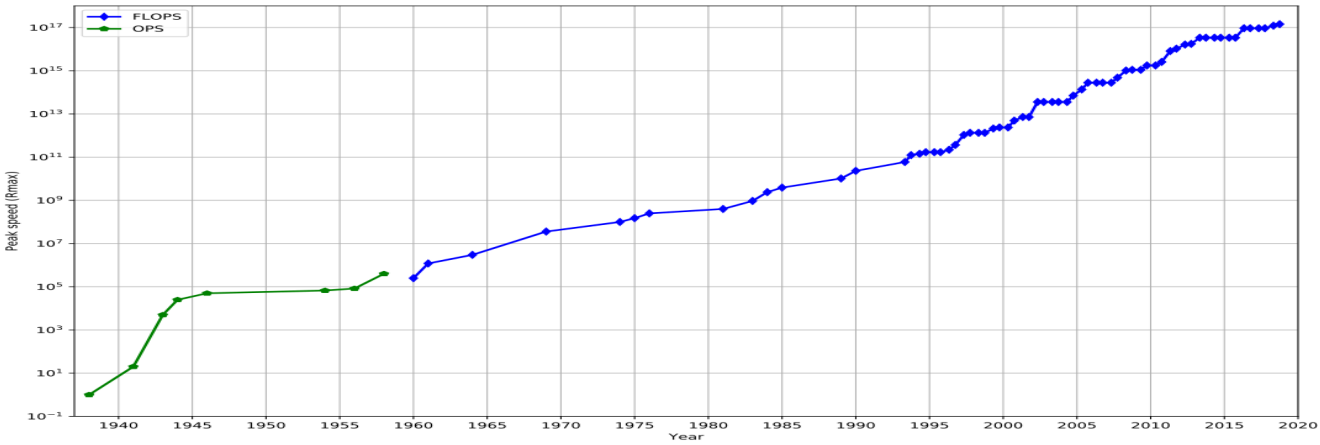
Python ist eine objektorientierte Programmiersprache, welche an Stelle von den herkömmlich üblicheren Klammerungen “{}” Einrückungen im Quelltext nutzt, um Schachtelungen im Code zu ordnen, was direkt eine bessere Übersichtlichkeit mit sich zieht. Weiterhin nutzt man in Python kein Semikolon um Zeilenabschlüsse anzuzeigen und aufgrund der dynamisch typisierten Variablen und Speicherverwaltung muss man nicht manuell den Typen deklarieren.

In [7]:
```
down = 0
up = 100
for i in range(1,10):
    guessed_age = int((up+down)/2)
    answer = input('Are you ' + str(guessed_age) + " years old?")
    if answer == 'correct':
        print("Nice")
        break
    elif answer == 'less':
        up = guessed_age
    elif answer == 'more':
        down = guessed_age
    else:
        print('wrong answer')
```

2. Was ist High Performance Computing?

In High Performance Computing können Datenverarbeitungen und Berechnungen in immensen Mengen ausgeführt werden. Hierbei unterscheidet sich die Struktur der Hard- und Software die genutzt wird stark von der eines herkömmlichen Copmuters. High Performance Computing genießt verschiedenste Anwendungsbereiche in Bereichen wie Wissenschaft, Business, Datenverarbeitung, Teilchenphysik oder Klima-Berechnungen. Es gibt physische Supercomputer und auch virtuelle Supercomputer, welche bei “Grid Computing” wie z.b “Jungle Computing”, also dem Verwenden von vielen separaten Systemen unter einem virtuellen Supercomputer verwendet werden.

Die Performance eines Supercomputers ist meist in “floating-point operations per second” gemessen (**FLOPS**). Hierzu ist in Abbildung 2 der Anstieg an maximal erreichten FLOPS unter Supercomputern der letzten 80 Jahre zu sehen.



[Abb. 2]
https://en.wikipedia.org/wiki/Supercomputer#/media/File:Supercomputing-rmax-graph2.svg

Unterscheidung

Anders als normale Copmuter sind Supercomputer nicht für den Allgemeingebrauch “weit” ausgelegt, sondern spezifisch für hohe Leistung konzipiert. High Performance Computer werden für Aufgaben benötigt, die “zu groß” für herkömmliche Computer wären, oder zu lange dauern würden.

Als Perspektive:

Beispielsweise kann ein moderner Laptop mit einem 3 Ghz Prozessor ca. 3 Milliarden Berechnungen pro Sekunde durchführen, während manche HPCs Billiarden von Berechnungen pro Sekunde durchführen können.

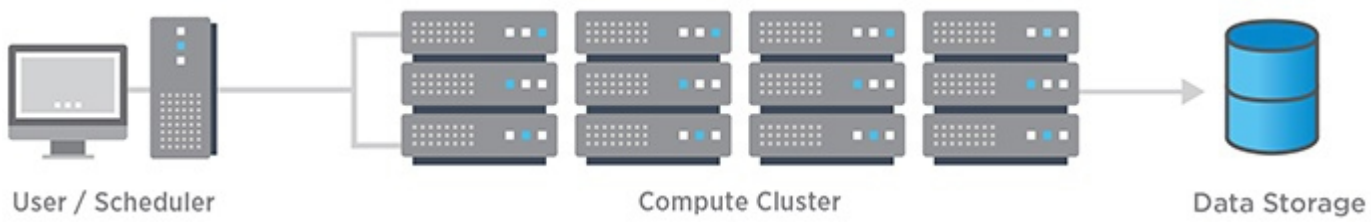
Nutzungsbereich

Wie zuvor angesprochen gibt es viele Nutzungsbereiche für High Performance Computer. Beispielsweise in der Wissenschaft werden HPCs für Problemlösung, Modellierung, Simulationen und Analysen von großen Datenmengen genutzt. Im Business-Bereich werden sie für Aufgaben wie Trend-Analysen in Clustern, Verarbeitung von Nutzerdaten zwecks Feature-Entwicklung und anderes genutzt. Im Bereich der Datenverarbeitung werden High Performance Copmuter in den Bereichen Smart Technologies, Verkehr, Internetnutzung, Klima und Medien genutzt.

Realisierung

High Performance Computer nutzen parallel arbeitende Algorithmen und Systeme um die immensen Mengen an Aufgaben und Daten zu bearbeiten. Hierbei werden Computer-Server in einem Cluster vernetzt,

welches mit dem Datenspeicher verbunden wird und dann von dem User angesprochen werden kann, wie es auch in Abbildung 3 grafisch zu sehen ist. Hierfür müssen alle Komponenten des HPCs in Höchst-Tempo funktioneren, da schon eine langsame Hardware-Komponente die Performance der Infrastruktur drastisch beeinflussen kann.



[Abb. 3] https://www.netapp.com/us/media/how-hpc-works.jpg

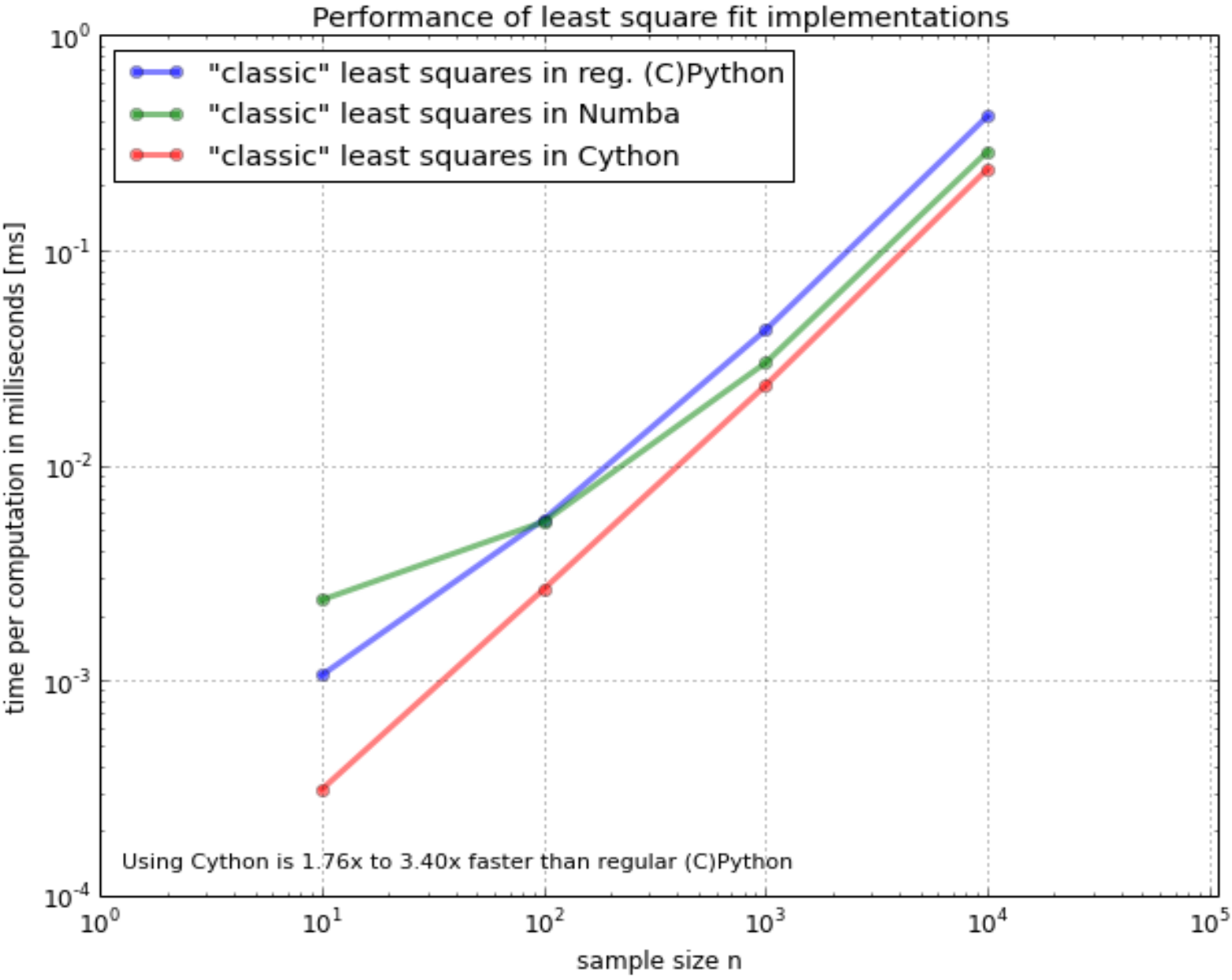
3. Wie wird Python in HPC realisiert?

Python wird mittels Nutzung von verschiedenen Libraries in High Performance Computing integriert. Unter diesen finden sich unter anderem Cython, NumPy & SciPy, welche zum Beispiel Tempo und Effizienz von Python Code erhöhen können. Unter Nutzung und Beachtung von Parallelität und Concurrency kann Code in Sachen Zeit und Effizienz weiter optimiert werden (siehe GIL).

Cython

Cython liefert die kombinierte Kraft von Python und C indem eine Moduldatei von Python in C konvertiert und kompiliert wird.

Diese Library wird während der Laufzeit kompiliert. Hierbei werden Schlüsselvariablen gesetzt, um den Compiler orientieren zu können. In Abbildung 4 ist der Voteil in einem Ersparnis von Zeit mittels Cython im Vergleich zu sehen.



[Abb. 4] https://i.stack.imgur.com/ZnCPh.png

NumPy und SciPy

NumPy und SciPy sind weitere der bereits genannten Libraries für Python. NumPy biete N-dimensionale Array-Objekte, sowie schnellere

Operationen auf Arrays für Python, nützliche Rechenoperationen der linearen Algebra und Fourir-Transformation um nur die größeren Vorteile zu nennen. Numpy eliminiert zeitlich “teure” explizite Schleifen und ersetzt sie durch spezielle NumPy-Operationen. Die Funktionen von NumPy nutzen hierbei effizienten C-Code um dies zu erreichen. SciPy enthält NumPy als Numerik-Basisbibliothek und zusätzliche Algorithmen zur numerischen Integration und Optimierung.

GIL

GIL steht für “Global Interpreter Lock”, welches in interpretierten Computersprachen verwendet wird, um Ausführung von Threads zu synchronisieren, sodass je nur ein nativer Thread ausgeführt werden kann. Ein Thread führt Python Code aus, während X andere schlafen, oder einen Input oder Output abwarten. Im Zusammenhang hiermit gibt es kooperatives und präventives Multitasking.
Kooperatives Multitasking bedeutet, dass sobald ein Thread schläft oder I/O abwartet, ein anderer Thread das GIL übernehmen und Python Code ausführen kann. Ein Python Thread kann jedoch "freiwillig" das Lock lösen, oder präventiv entzogen bekommen.
Präventives Multitasking bedeutet, dass ein Thread bei ununterbrochener Laufzeit (1000 bytecode instrucions in Python 2, oder 15 Millisekunden Laufzeit in Python 3) das Lock abgibt und ein anderer Thread durchlaufen kann.

Thread Safety

In Python wird Text in ein simpleres Binärformat namens Bytecode compiliert und diese Anweisungen werden dann vom interpreter durchgeführt. Ein Problem dabei ist, dass während der Laufzeit das GIL

periodisch abgegeben wird, ohne den Thread nach “Erlaubnis” zu fragen. In Python 2 geschieht dies alle 1000 bytecode Anweisungen und in Python 3 alle 15 millisekunden.
Wenn ein Thread also das GIL technisch an jedem Punkt verlieren kann, muss man die sogenannte Thread-Safety beachten. In Python sind zwar viele Operationen atomar, aber nicht alle. Nehmen wir als Beispiele += und sort().
Beispiel:

```
n = 0

def foo():
    global n
    n += 1
```

[Abb. 5] https://opensource.com/article/17/4/grok-gil

Mittels Python's **dis** Modul können wir den Bytecode des in Abbildung 5 gezeigten Codes in Abbildung 6 ansehen:

```
>>> import dis
>>> dis.dis(foo)
LOAD_GLOBAL              0 (n)
LOAD_CONST               1 (1)
INPLACE_ADD
STORE_GLOBAL             0 (n)
```

[Abb. 6] https://opensource.com/article/17/4/grok-gil

Die Zeile n += 1 wurde in vier bytecodes kompiliert:

1. Lade den Wert von n auf den Stack
2. Lade die Konstante 1 auf den Stack
3. Summiere die beiden Werte auf dem Stack

4. Speichere die Summe in n.

Da bspw. in Python 2 alle 1000 bytecodes ein Thread das Lock präventiv verlieren kann, wäre es hier möglich, dass dies zwischen dem Laden von n auf den Stack und dem Speichern der Summe in n geschehen könnte, was ein Problem darstellt

Beispiel für solche Fehler:

```
threads = []
for i in range(100):
    t = threading.Thread(target=foo)
    threads.append(t)

for t in threads:
    t.start()

for t in threads:
    t.join()

print(n)
```

[Abb. 7] https://opensource.com/article/17/4/grok-gil

Normalerweise gibt der Code in Abbildung 7 100 aus, aber manchmal kommen 98 oder 99 raus, wenn das Update eines Threads von einem anderen überschrieben wurde.

Hier bräuchten wir also trotz GIL ein Lock um gemeinsam veränderbare Zustände zu schützen

Beispiel an einer atomaren Operatoin; sort():

```
lst = [4, 1, 3, 2]

def foo():
    lst.sort()
```

[Abb. 8] https://opensource.com/article/17/4/grok-gil

Wenn wir den Code aus Abbildung 8 als Bytecode ansehen, sehen wir, dass der Bytecode in Abbildung 9 zeigt , dass **sort()** nicht unterbrochen werden kann, da es atomar ist:

```
>>> dis.dis(foo)
LOAD_GLOBAL              0 (lst)
LOAD_ATTR                1 (sort)
CALL_FUNCTION            0
```

[Abb. 9] https://opensource.com/article/17/4/grok-gil

Die Anweisungen sind gemäß Abbildung 9 wie folgt:

1. Lade den Wert von 1st auf den Stack
2. Lade die sort() Methode auf den Stack
3. Rufe die sort() Methode auf.

Die Operation "+=" wirkt auf den ersten blick simpler als "sort()", ist jedoch nicht atomar, während "sort()" eine solche atomare Operation ist. Da der Aufruf an "sort()" selbst ein einzelner bytecode ist, gibt es keine Möglichkeit für den Thread, das Lock während des Aufrufs entzogen zu bekommen. Deshalb sollten man jedoch nicht selbst prüfen, welche Operationen alle atomar sind, sondern immer beim Auslesen und Schreiben auf geteilte veränderbare Zustände ein Lock verwenden.

Anders als in Java können wir die Locks jedoch simpel halten, da mehrere Threads nicht parallel Python Code ausführen können.

Concurrency

Code, der viele Netzwerk-Operationen abwarten muss, profitiert von mehreren Threads, obwohl nur ein Thread zur Zeit Python Code

ausführen kann (**Concurrency**).

```
import threading
import requests

urls = [...]

def worker():
    while True:
        try:
            url = urls.pop()
        except IndexError:
            break  # Done.

        requests.get(url)

for _ in range(10):
    t = threading.Thread(target=worker)
    t.start()
```

[Abb. 10] https://opensource.com/article/17/4/grok-gil

Wie wir bereits wissen, geben Threads das GIL ab, während sie auf die Rückgabe einer URL warten. Somit können in dem in Abbildung 10 gezeigten Code, mehrere Threads auf die Übertragung warten, während das GIL weitergegeben wird um auf einem anderen Thread den weiteren Python Code auszuführen.

Parallelism

Aufgrund des GIL ist Parallelität innerhalb von Threads nicht möglich. Jedoch kann man mehrere Prozesse nutzen, was zwar mehr Speicher einnimmt, aber auch mehrere CPUs besser nutzen kann. Das hier gegebene Beispiel in Abbildung 11 läuft schneller, durch Aufteilung in 10 parallele Prozesse über mehrere Kerne verteilt, wohingegen 10 Threads deutlich langsamer wären, da immer nur ein Thread Python Code ausführen kann. Jeder Prozess besitzt ein eigenes GIL, weshalb diese parallel durchgeführt werden können.

```
import os
import sys

nums =[1 for _ in range(1000000)]
chunk_size = len(nums) // 10
readers = []

while nums:
    chunk, nums = nums[:chunk_size], nums[chunk_size:]
    reader, writer = os.pipe()
    if os.fork():
        readers.append(reader)  # Parent.
    else:
        subtotal = 0
        for i in chunk: # Intentionally slow code.
            subtotal += i

        print('subtotal %d' % subtotal)
        os.write(writer, str(subtotal).encode())
        sys.exit(0)

# Parent.
total = 0
for reader in readers:
    subtotal = int(os.read(reader, 1000).decode())
    total += subtotal

print("Total: %d" % total)
```

[Abb. 11] https://opensource.com/article/17/4/grok-gil

4. WARUM PYTHON IN HPC?

Pro

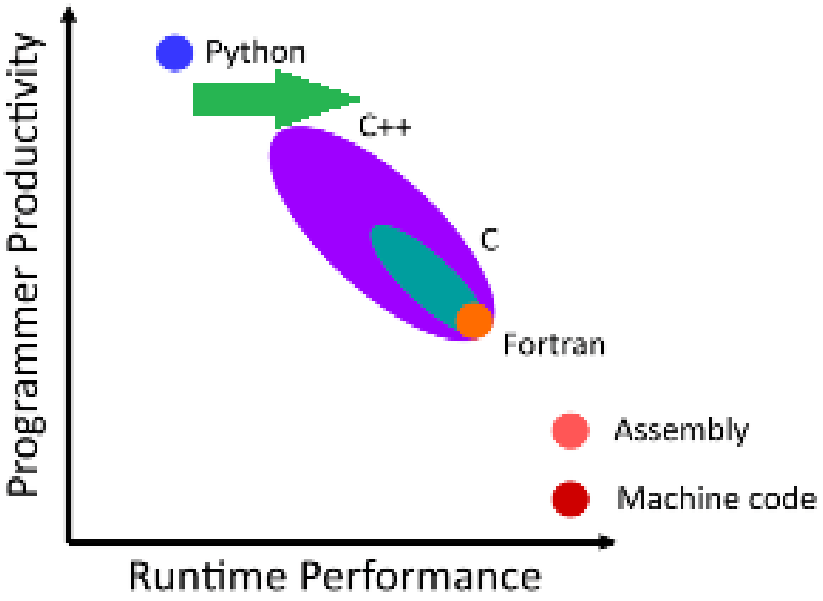
Python Code kann mächtige Konzepte in wenig Zeilen ausdrücken und mittels einer großen Zahl an einfach aufrufbaren Python libraries wie NumPy, Cython etc. kann Python Code sehr effizient und optimiert für High Performance Computing gestaltet werden. Weiterhin kann Python durch Beachtung und Nutzung von Concurrency und Parallelism sehr effizient für HPC-Aufgaben optimiert werden.

Contra

Python Code ist im Verhältnis recht langsam. Beispielsweise "For"-Loops in Python sind sehr zeitintensiv. Ebenso sorgen die dynamische Typisierung und die Tatsache, dass Python eine interpretierte Sprache ist für hohe Geschwindigkeitseinbußen. Dies mag bei normalen Desktop-Programmen nicht all zu gravierend sein, wird im Bereich des High Performance Computing jedoch problematisch. Weiterhin ist die Nutzung von Python im High Performance Parallel Rechnen relativ neu, wodurch noch verhältnismäßig wenige "parallel processing" Techniken bekannt sind.

Differenzierung

Standard Python Code ist um vielfaches (ca. 180x) langsamer als äquivalenter Code in C, C++ oder Fortran, jedoch kann das Tempo mittels NumPy, SciPy etc. angeglichen werden (nur noch 6x). Python-Code wird unter Beachtung von Concurrency und Parallelität weiter beschleunigt & zeitlich effizienter, was die Sprache im Rahmen des High Performance Computing weiter nach vorne setzt. So können Pythons Stärken und Funktionen mit angemessener Zeiteffizienz genutzt werden. In Abbildung 12 ist zu sehen, wie die hohe Produktivität von Python mittels dieser Techniken auf der "X-Achse", also in Sachen Performance weiter nach vorn gebracht werden kann.



[Abb. 12] https://www.researchgate.net/figure/Code-Languages-Top-Feature-Comparison-in-the-Case-of-Numerical-Simulations_fig4_336577121

5. FAZIT

Python ist leicht verständlich und macht HPC somit zugänglicher und mit kombinierten Paketen können Effizienz und Tempo an andere Programmiersprachen im Feld angepasst werden. Simulationen und Analysen können somit effizient und parallel berechnet werden. Thread-Safety, sowie Concurrency und Parallelität sind jedoch wichtig, um Python in HPC noch effektiver nutzen zu können und müssen von Programmierern beachtet werden. Mit angepasstem Tempo können die Stärken von Python auch in HPC zur Geltung kommen

6. Quellen

https://dl.acm.org/doi/10.1109/MCSE.2007.58

https://www.csc.fi/documents/200270/224366/basic_python.pdf/e18a3c4e-fe61-421d-bc10-6bfefcc23639

https://doi.org/10.1109/MCSE.2007.58

https://www.netapp.com/us/info/what-is-high-performance-computing.aspx

https://www.uibk.ac.at/austriangrid/manuals/galileocomputing_python.pdf

http://www.inztitut.de/blog/glossar/python/

https://www.python.org/

https://cython.org/

https://scipy.org/

http://numpy.scipy.org/

http://www.inztitut.de/blog/glossar/numpy/

https://opensource.com/article/17/4/grok-gil

https://wiki.python.org/moin/GlobalInterpreterLock