Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Python in HPC

MALTE EICKHOFF

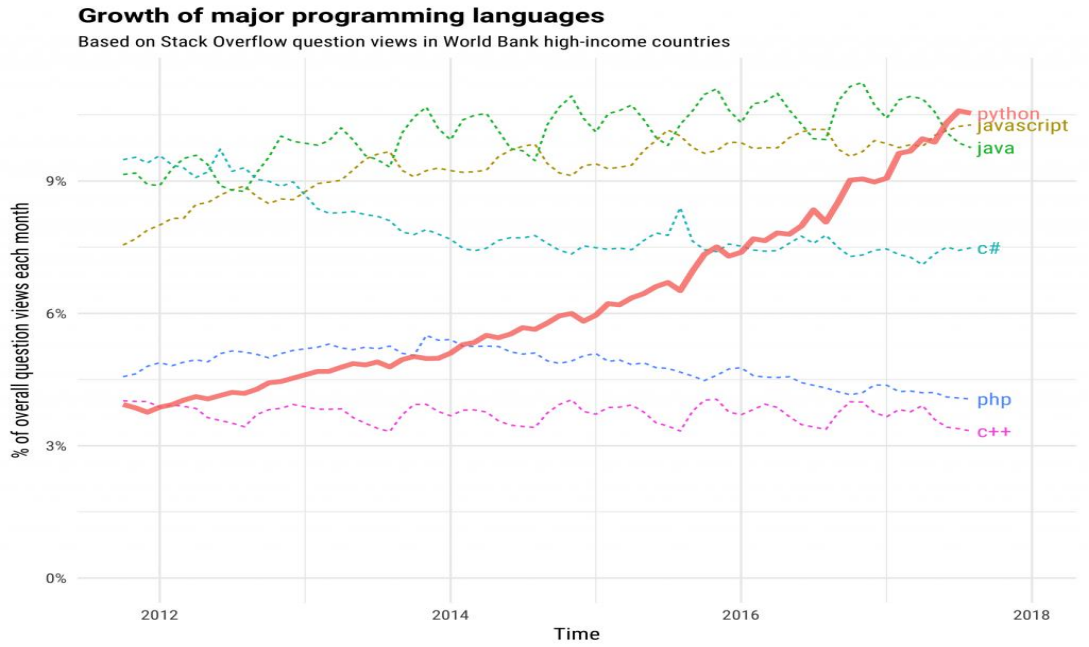
SOFTWAREENTWICKLUNG IN DER WISSENSCHAFT

Gliederung:

1. Was ist Python?
2. Was ist High Performance Computing?
3. Wie wird Python in HPC realisiert?
4. Warum Python in HPC?
5. Fazit
6. Quellen

1. Was ist Python? - Einführung

- hochgradig dynamisch typisierte, universelle Programmiersprache
- Genießt große Popularität in den Bereichen von künstlicher Intelligenz und Deep Learning



https://stackoverflow.blog/2017/09/06/incredible-growth-python/

1. Was ist Python? - Pro & Contra

Pro:

- Saubere und einfache Programmiersprache
- Leicht lesbar und intuitiv gestaltet
- Dynamisch typisierte Variablen
- Automatische Speicherverwaltung

Contra:

- Langsame Kompilierungszeiten
- Durch dezentrale Organisation von Python durch Pakete etc. kann der Neueinstieg schwer fallen

1. Was ist Python? - Syntax

- Objektorientierte Programmiersprache
- Einrückungen im Quellcode an Stelle von "{}" und ";"
- Dynamisch typisierte Variablen und Speicherverwaltung

In [7]:

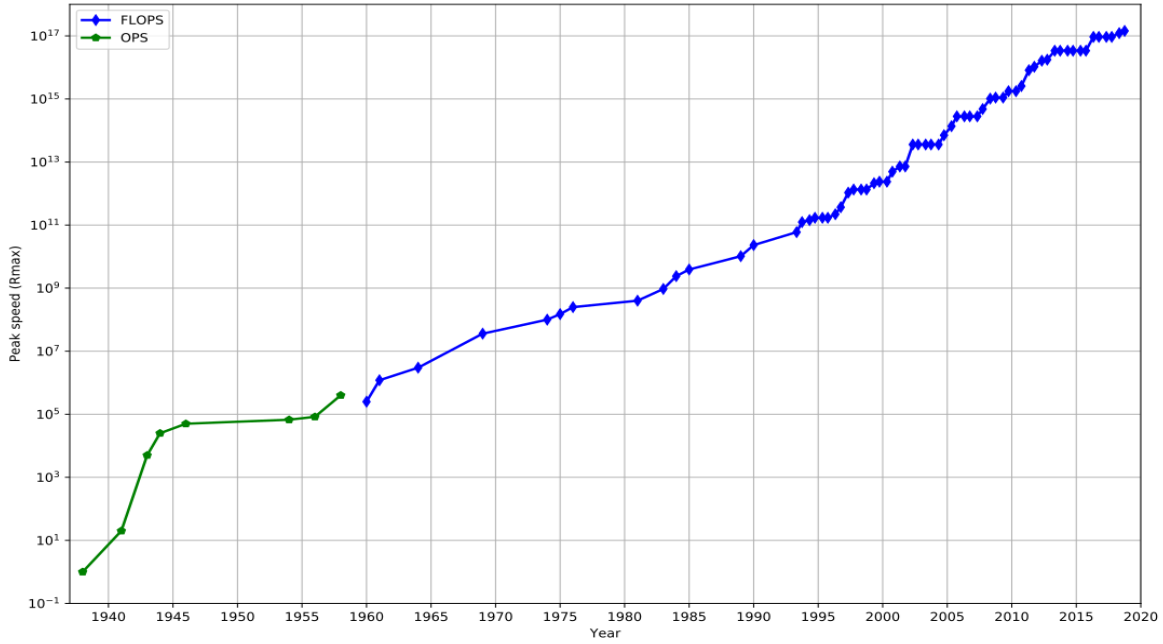
```
1  down = 0
2  up = 100
3  for i in range(1,10):
4      guessed_age = int((up+down)/2)
5      answer = input('Are you ' + str(guessed_age) + " years old?")
6      if answer == 'correct':
7          print("Nice")
8          break
9      elif answer == 'less':
10         up = guessed_age
11     elif answer == 'more':
12         down = guessed_age
13     else:
14         print('wrong answer')
```

2. Was ist High Performance Computing?

- In High-Performance Computing können Datenverarbeitungen und Berechnungen in immensen Mengen ausgeführt werden
- **Struktur** der Hard- und Software unterscheiden sich von herkömmlichen Computern
- Verschiedenste Anwendungsbereiche (Wissenschaft, Business, Datenverarbeitung, Teilchenphysik, Wetter- & Klima-Berechnungen)
- Es gibt **physische** Supercomputer und **virtuelle**, welche bei "Grid Computing" wie z.B. "Jungle Computing", also dem Verwenden von vielen separaten Systemen unter einem virtuellen Supercomputer, verwendet werden

2. Was ist High Performance Computing?

Die Performance eines Supercomputers ist meist in "floating-point operations per second" gemessen (**FLOPS**)



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

2. Was ist HPC? - Unterscheidung

- Anders als normale Computer 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

➢ <u>Als Perspektive:</u>

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

2. Was ist HPC? - Nutzungsbereich

- **Wissenschaft**

➢ Problemlösung, Modellierung, Simulationen und Analysen von großen Datenmengen

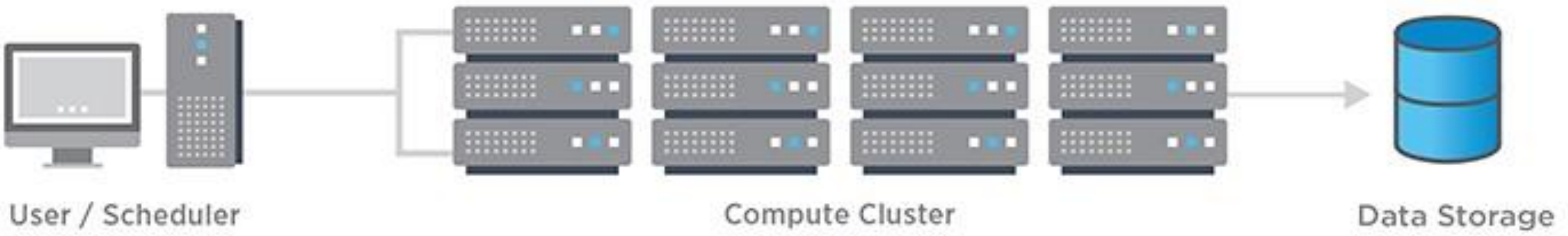
- **Business**

➢ Trend-Analysen in Clustern, Verarbeitung von Nutzerdaten zwecks Feature-Entwicklung

- **Datenverarbeitung** (Smart Technologies, Verkehr, Internetnutzung, Klima, Medien)

2. Was ist HPC? - Realisierung

- HPCs nutzen parallel arbeitende Algorithmen und Systeme
- Computer-Server werden in einem Cluster vernetzt
- Cluster wird mit Datenspeicher verbunden und von User angesprochen
- Alle Komponenten des HPCs müssen in Höchst-Tempo funktionieren, da schon eine langsame Komponente die Performance der Infrastruktur drastisch beeinflussen kann

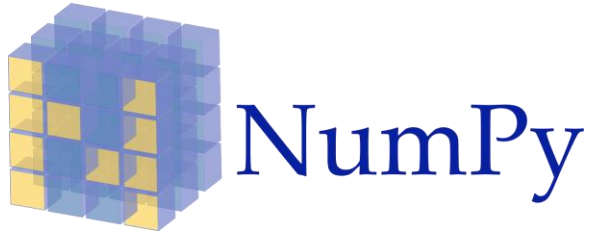


https://www.netapp.com/us/media/how-hpc-works.jpg

3. Wie wird Python in HPC realisiert?

- Python wird mittels Nutzung von verschiedenen Libraries integriert

- Unter diesen finden sich unter anderem **Cython**, **NumPy** & **SciPy**
- ➢ Diese erhöhen z.B. Tempo und Effizienz von Python Code



- Mittels Parallelität und Concurrency kann Code zeitlich und effizienstechnisch weiter optimiert werden (siehe **GIL**)

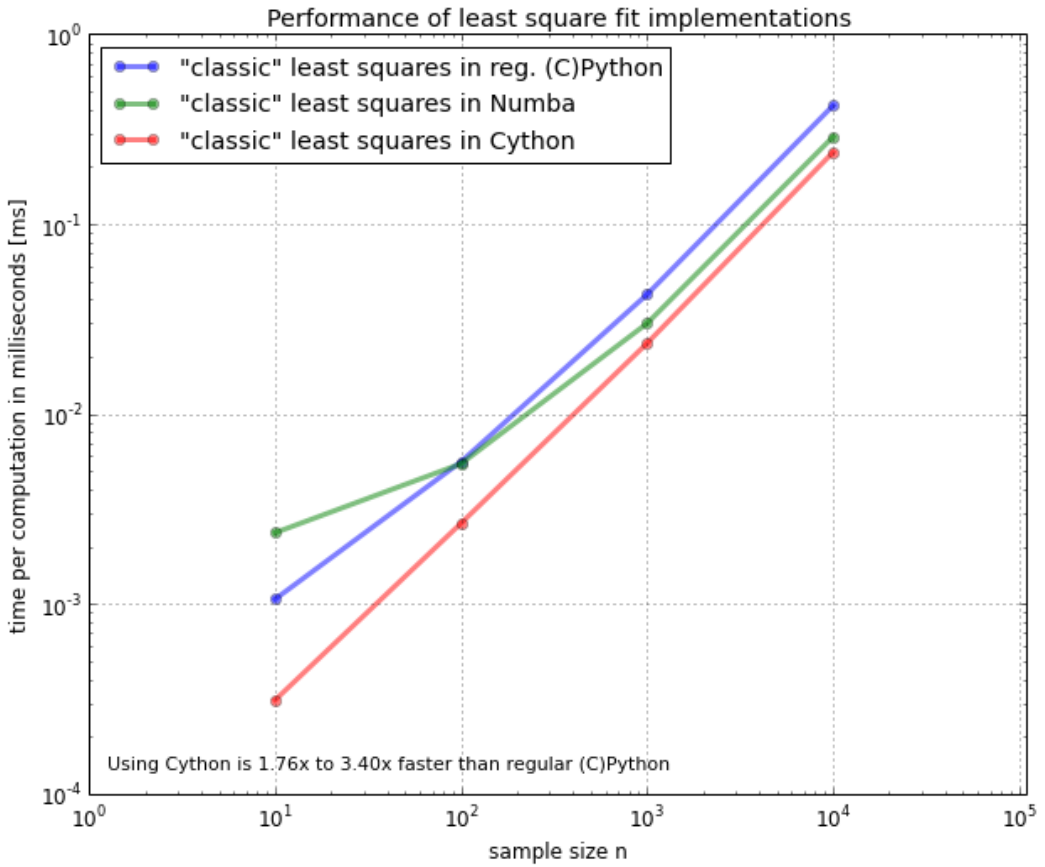
3. Python in HPC - Cython

- Cython liefert die kombinierte Kraft von Python und C

- Python Moduldatei wird in C konvertiert und kompiliert

➢ Diese Library wird während der Laufzeit kompiliert

- Schlüsselvariablen werden gesetzt, um den Compiler zu orientieren



https://i.stack.imgur.com/ZnCPh.png

3. Python in HPC - NumPy und SciPy

- NumPy bietet N-dimensionale Array-Objekte, sowie schnellere Operationen auf Arrays
- Nützliche Rechenoperationen der linearen Algebra, Fourier-Transformation und weitere
- Eliminiert zeitlich "teure" explizite Schleifen und ersetzt sie durch NumPy Operationen
- NumPy Funktionen nutzen effizienten C-Code

- SciPy enthält NumPy als Numerik-Basisbibliothek
- Enthält zusätzlich Algorithmen zur numerischen Integration und Optimierung

3. Python in HPC - GIL

- GIL steht für **Global Interpreter Lock**

➢ Wird in interpretierten Computersprachen verwendet, 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/Output abwarten

- **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 "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 durch laufen kann

3. Python in HPC - Thread-Safety

- In Python wird Text in ein simpleres Binärformat namens Bytecode compiliert und diese Anweisungen werden dann vom interpreter durchgeführt
- **Problem**: Während der Laufzeit wird das GIL periodisch abgegeben, ohne den Thread nach "Erlaubnis" zu fragen

➢ Alle 1000 bytecode Anweisungen (Python 2), oder 15 millisekunden (Python 3)

- siehe präventives Multitasking

- Wenn ein Thread 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()**

3. Python in HPC - Thread-Safety

Beispiel:

```
n = 0

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

Mittels Python's **dis** Modul können wir den Bytecode ansehen:

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

3. Python in HPC - Thread-Safety

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

➢ Lade den Wert von n auf den Stack

➢ Lade die Konstante 1 auf den Stack

➢ Summiere die beiden Werte auf dem Stack

➢ 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

3. Python in HPC - Thread-Safety

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)
```

Normalerweise gibt dieser Code 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

3. Python in HPC - Thread-Safety

Beispiel an einer atomare Operation; **sort()**:

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

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

Der Bytecode zeigt hier, dass **sort()** nicht unterbrochen werden kann, da es atomar ist:

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

- Lade den Wert von **1st** auf den Stack
- Lade die **sort()** Methode auf den Stack
- Rufe die **sort()** Methode auf.

3. Python in HPC - Thread-Safety

- **+=** wirkt simpler als **sort()**, ist jedoch nicht atomar, während **sort()** dies 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 kann.

3. Python in HPC - 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()
```

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

3. Python in HPC - 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 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

```python
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)
```

4. Warum Python in HPC? - Pro

- Python Code kann mächtige Konzepte in wenig Zeilen ausdrücken
- Große Zahl an einfach aufrufbaren Python libraries
- ➢ Mittels NumPy, Cython etc. kann Python Code sehr effizient und optimiert für HPC gestaltet werden
- Durch Nutzung von Concurrency und Parallelism kann Python sehr effizient für HPC-Aufgaben optimiert werden

4. Warum Python in HPC? - Contra

- Langsam (bspw. For-loops in Python sind sehr Zeit-intensiv)

- Dynamische Typisierung und dass Python eine interpretierte Sprache ist sorgen für hohe Geschwindigkeits-Einbußen

➢ Bei normalen Desktop Programmen nicht so gravierend, aber bei HPC problematisch

- Python Nutzung im HPC parallel rechnen ist relativ neu, wodurch noch verhältnismäßig weniger "parallel processing" Techniken bekannt sind

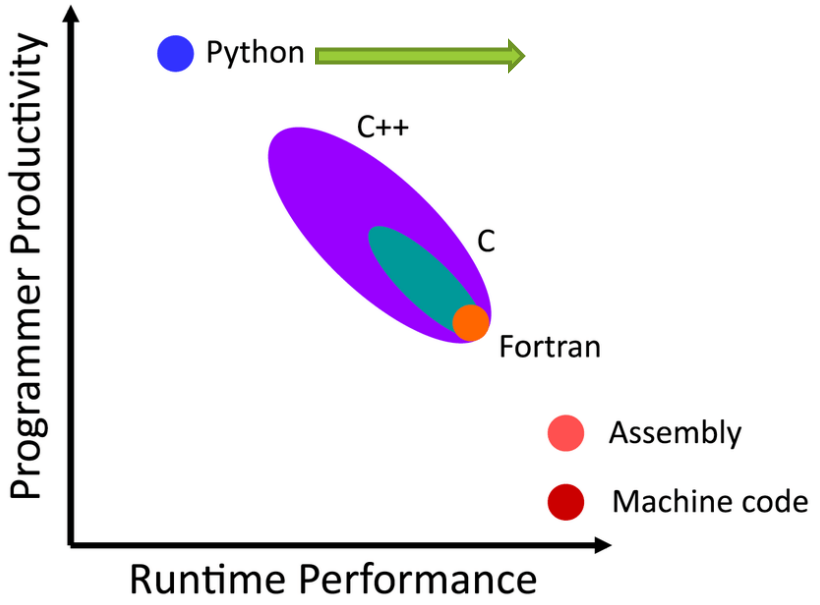
4. Warum Python in HPC? - Differenzierung

- Standard Python Code ist um vielfaches (ca. 180x) langsamer als äquivalenter Code in C, C++ oder Fortran
- Tempo kann mittels NumPy, SciPy etc. angeglichen werden (nur noch 6x)
- Cython, nuitka, u.ä. machen Python Programme zu kompilierten C Programmen

- Code wird unter Beachtung von Concurrency und Parallelität weiter beschleunigt & effizienter

➢ So können Pythons Stärken und Funktionen mit angemessener Zeiteffizienz genutzt werden

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
- Mit kombinierten Paketen können Effizienz und Tempo angepasst werden

➢ Simulationen und Analysen können effizient und parallel berechnet werden

- Thread-Safety, sowie Concurrency und Parallelität sind wichtig, um Python in HPC noch effektiver nutzen zu können
- 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/

6. Quellen

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