Multithreading und Multiprocessing in Python

Proseminar: Python im Hochleistungsrechnen

Pascal Lindner

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2019-24-06

informatik die zukunft

Gliederung

1 Multithreading

2 GIL

3 Multiprocessing

4 Zusammenfassung

5 Literatur

Multithreading - Wieso?

- Performance Gewinn

Multithreading - Wieso?

- Performance Gewinn
- Modulare (Hochleistungs-) Rechner

Multithreading - Wieso?

- Performance Gewinn
- Modulare (Hochleistungs-) Rechner
- Paralleles Rechnen

Multithreading - Wieso?

- Performance Gewinn
- Modulare (Hochleistungs-) Rechner
- Paralleles Rechnen
- I/O-abhängige Programme

Rechnung ohne paralleles Rechnen:

- $(A * B) + (C * D)$

Rechnung ohne paralleles Rechnen:

- $(A * B) + (C * D) \qquad //(A * B) = X$

Rechnung ohne paralleles Rechnen:

- $(A * B) + (C * D) \qquad //(A * B) = X$
- $\Rightarrow X + (C * D)$

Rechnung ohne paralleles Rechnen:

- $(A * B) + (C * D) \qquad //(A * B) = X$
- $\Rightarrow X + (C * D) \qquad //(C * D) = Y$

Rechnung ohne paralleles Rechnen:

- $(A * B) + (C * D) \qquad //(A * B) = X$
- $\Rightarrow X + (C * D) \qquad //(C * D) = Y$
- $\Rightarrow X + Y$

Rechnung ohne paralleles Rechnen:

- $(A * B) + (C * D)$ $//(A * B) = X$
- $\Rightarrow X + (C * D)$ $//(C * D) = Y$
- $\Rightarrow X + Y$ $//X + Y =$ Ergebnis

Rechnung mit parallelem Rechnen:

$$(A * B) + (C * D)$$

Rechnung mit parallelem Rechnen:

$$(A * B) + (C * D)$$

$$A * B = X \qquad C * D = Y$$

Rechnung mit parallelem Rechnen:

$$(A * B) + (C * D)$$

$$A * B = X \qquad C * D = Y$$

$$X + Y$$

Aufbau Von-Neumann Rechner

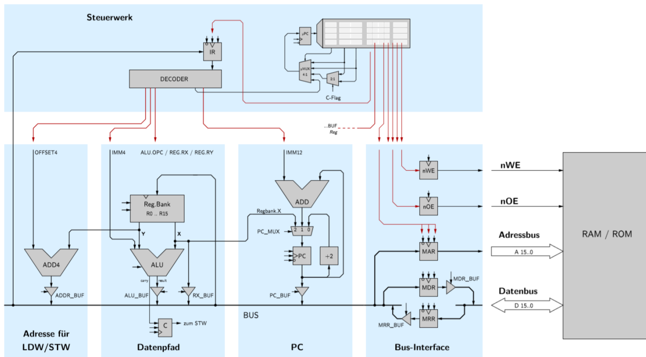

Abbildung: Aufbau eines Von-Neumann Rechners

Multithreading in Python

- from threading import Thread
- Konstruktor:
  Thread(target=None, name=None, args=(), kwargs={}, *, daemon=None)

Multithreading in Python

- start()

Multithreading in Python

- start()
- run()

Multithreading in Python

- start()
- run()
- join(timeout=None)

Multithreading in Python

- start()
- run()
- join(timeout=None)
- is_alive()

Einfaches Python Programm mit einem Thread

```
import time

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

start = time.time()
countdown(COUNT)
end = time.time()

print('Time taken in seconds: ', end - start)
```

Einfaches Python Programm mit zwei Threads

```
import time
from threading import Thread

COUNT = 50000000
def countdown(n):
    while n>0:
        n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))
t2 = Thread(target=countdown, args=(COUNT//2,))
start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()
print('Time taken in seconds: ', end - start)
```

Auswertung

- Zeit für Single-threaded: 4,47s
- Zeit für Multi-threaded: 8,19s

Probleme beim Multithreading

- Rechnung auf Maschinenebene:

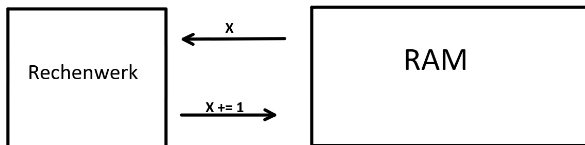

Abbildung: Inkrementieren von x

Probleme beim Multithreading

- Rechnung auf Maschinenebene mit 2 CPU-Kernen:

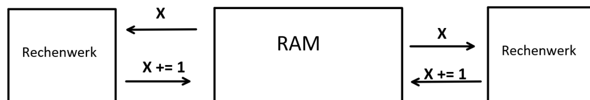

Abbildung: Es steht x+1 im Speicher, obwohl zweimal inkrementiert wurde

Probleme und Gefahren

- Besonders problematisch für Reference-Counting
  - Objekte können gelöscht werden obwohl noch Referenzen existieren
  - Speicher wird „zugemüllt“
- C Extensions verlassen sich häufig auf Thread-Safety

Locks

- Locks können Objekte schützen

Locks

- Locks können Objekte schützen
- from threading import Lock

Locks

- Locks können Objekte schützen
- from threading import Lock
- Zwei Zustände: locked und unlocked

Locks

- Locks können Objekte schützen
- from threading import Lock
- Zwei Zustände: locked und unlocked
- acquire(blocking=True, timeout=-1)

Locks

- Locks können Objekte schützen
- from threading import Lock
- Zwei Zustände: locked und unlocked
- acquire(blocking=True, timeout=-1)
- release()

Einfaches Python Programm mit zwei Threads

```
import time
from threading import Thread

COUNT = 50000000
def countdown(n):
    while n>0:
        n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))
t2 = Thread(target=countdown, args=(COUNT//2,))
start = time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end = time.time()
print('Time taken in seconds: ', end - start)
```

Jetzt mit Locks

```
1  from threading import Thread, Lock
2
3  count = 50000000
4  def countdown(l):
5      global count
6      while count>0:
7          l.acquire()
8          count -= 1
9          l.release()
10
11 lock = Lock()
12 t1 = Thread(target=countdown, args=(lock,))
13 t2 = Thread(target=countdown, args=(lock,))
14 t1.start()
15 t2.start()
16 t1.join()
17 t2.join()
```

DeadLocks

DeadLocks

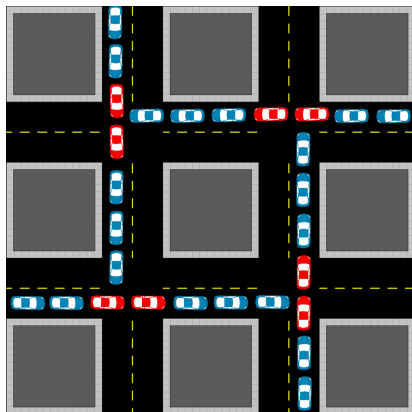

Abbildung: Deadlock: https://hackernoon.com/how-to-avoid-a-deadlock-while-writing-concurrent-programs-java-example-988bb07db25f

GIL

- Global Interpreter Lock
  - Ein Lock für den gesamten Interpreter
- Garantiert Thread-Safety für C-Extensions
- Multithreading für CPU-lastige Programme nicht möglich

GIL - Funktionsweise

- Unterscheidet zwischen I/O-Programmen und CPU-lastigen Programmen
- GIL wird nach jedem I/O freigegeben
- z.B. beim Lesen, Schreiben, Datenbankzugriffen, usw.

GIL - Funktionsweise

- Unterscheidet zwischen I/O-Programmen und CPU-lastigen Programmen
- GIL wird nach jedem I/O freigegeben
- z.B. beim Lesen, Schreiben, Datenbankzugriffen, usw.

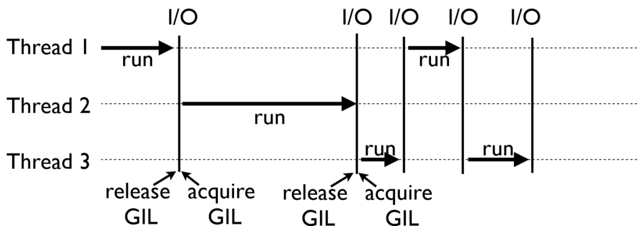

Abbildung:
https://www.dabeaz.com/python/UnderstandingGIL.pdf

GIL - Funktionsweise

- Bei CPU-abhängigen Programmen nach 100 Ticks freigegeben
- 1 Tick entspricht einem Interpreter-Befehl
- Lässt sich durch sys.setcheckinterval() anpassen

GIL - Funktionsweise

- Bei CPU-abhängigen Programmen nach 100 Ticks freigegeben
- 1 Tick entspricht einem Interpreter-Befehl
- Lässt sich durch sys.setcheckinterval() anpassen

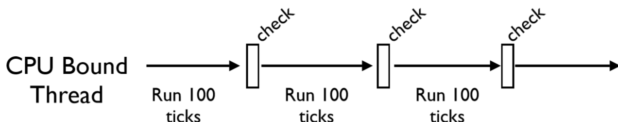

Abbildung:
https://www.dabeaz.com/python/UnderstandingGIL.pdf

GIL - Funktionsweise

- Dabei kämpfen die Threads um das GIL
- Kann dazu führen, dass Threads ungleichmäßig arbeiten

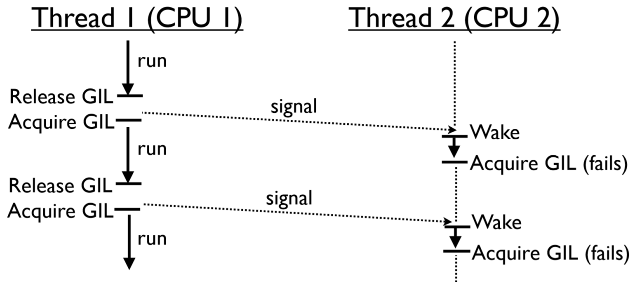

Abbildung:
https://www.dabeaz.com/python/UnderstandingGIL.pdf

GIL

- Sehr leicht zu implementieren
- Hohe Performance für Programme mit einem Thread
- Erlaubt einfache Benutzung von C/C++ Extensions

Alternativen?

- GIL ist umstritten

Alternativen?

- GIL ist umstritten
- „I'd welcome a set of patches into Py3k only if the performance for a single-threaded program (and for a multi-threaded but I/O-bound program) does not decrease.“
  - Guido van Rossum

Alternativen?

- GIL ist umstritten
- „I'd welcome a set of patches into Py3k only if the performance for a single-threaded program (and for a multi-threaded but I/O-bound program) does not decrease.“
  - Guido van Rossum
- Andere Implementation

Alternativen?

- GIL ist umstritten
- „I'd welcome a set of patches into Py3k only if the performance for a single-threaded program (and for a multi-threaded but I/O-bound program) does not decrease.“
  - Guido van Rossum
- Andere Implementation
- Gilectomy

Multiprocessing

- Prozesse mit eigenem Interpreter
- Teilen sich nicht den Arbeitsspeicher
- Etwas CPU-lastiger als Threads

API

- Sehr ähnlich zu threading
- from multiprocessing import Process
- Konstruktor:
  Process(target=None, name=None, args=(), kwargs={}, *, daemon=None)

Wichtigste Methoden

- start()

Wichtigste Methoden

- start()
- run()

Wichtigste Methoden

- start()
- run()
- join(timeout=None)

Wichtigste Methoden

- start()
- run()
- join(timeout=None)
- is_alive()

Einfaches Python Programm mit zwei Prozessen

```
1  from multiprocessing import Process
2  import time
3
4  COUNT = 50000000
5  def countdown(n):
6      while n>0:
7          n -= 1
8
9  if __name__ == '__main__':
10     p1 = Process(target=countdown, args=(COUNT//2,))
11     p2 = Process(target=countdown, args=(COUNT//2,))
12     start = time.time()
13     p1.start()
14     p2.start()
15     p1.join()
16     p2.join()
17     end = time.time()
18     print('Time taken in seconds: ', end - start)
```

Auswertung

- Zeit für Multi-process: 2,70s

Auswertung

- Zeit für Multi-process: 2,70s
- Zeit für Single-threaded: 4,47s
- Zeit für Multi-threaded: 8,19s

Prozess-Pools

- Prozess-Pools
- Verwaltet Arbeiter-Prozesse
- Etwas effizienter bei sehr vielen Aufgaben

Prozess-Pools

- apply_async(func, args)
- map(func, iterable)
- map_async(func. iterable)

Async-Result

- AsyncResult wird von apply_async und map_async zurückgegeben
- get(timeout)
- wait(timeout)
- ready()
- successfull()

Prozess-Pools

```
1  from multiprocessing import Pool
2  import time
3
4  COUNT = 50000000
5  def countdown(n):
6      while n>0:
7          n -= 1
8
9  if __name__ == '__main__':
10     start = time.time()
11     with Pool(2) as p:
12         p.map(countdown, [COUNT//2, COUNT//2])
13     end = time.time()
14     print('Time taken in seconds: ', end - start)
```

Zusammenfassung

- Multithreading in Python möglich
  - Performancegewinn nur bei I/O-Programmen
  - GIL verhindert Multithreading für CPU-lastige Programme
- Multiprocessing
  - Ähnlich zu Multithreading
  - Umgeht das GIL mit eigenem Interpreter
  - Sehr hohe Performance für anspruchsvolle Rechnungen

Literatur

- https://docs.python.org/2/library/multiprocessing.html
- https://docs.python.org/3/library/threading.html#threading.Thread
- https://realpython.com/python-gil/
- https://www.artima.com/weblogs/viewpost.jsp?thread=214235
- https://lwn.net/Articles/754577/
- http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/026_c_paralleles_rechnen_003.htm
- https://tams.informatik.uni-hamburg.de/lectures/2018ws/vorlesung/rs/
- https://www.dabeaz.com/python/UnderstandingGIL.pdf