Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

Multithreading und Multiprocessing in Python

vorgelegt von

Pascal Lindner

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

Studiengang: Informatik
Matrikelnummer: 6950489

Hamburg, 2019-08-01

Contents

1 Multithreading

Bevor wir uns mit Multithreading in Python speziell beschäftigen, schauen wir uns zunächst an, wozu Multithreading eigentlich nützlich ist. Das Ziel beim Multithreading ist in erster Linie Performancegewinn. Es gibt einige Möglichkeiten, die Performance von Programmen in Python zu erhöhen. Durch das Verwenden von C-Extensions, NumPy und anderen Software-bezogenen Tools lässt sich die Geschwindigkeit von Python Programmen bereits deutlich erhöhen, aber das Ziel von Multithreading ist es, die Hardware des Rechners besser zu nutzen. Heutzutage besitzt jeder Rechner mehrere Prozessorkerne und selbst Hochleistungsrechner besitzen meist nicht bessere oder leistungsstärkere Hardware, sondern einfach mehr Hardware. Diese Hardware ist modular aufgebaut, also lässt sich die Leistung eines solchen Rechners nicht durch linearen Programmcode effizient ausnutzen. Beim Multithreading werden die Aufgaben des Programms auf mehrere Threads aufgeteilt, die dann von unterschiedlichen Prozessorkernen ausgeführt werden können.

Dabei lassen sich Programme grob in 2 Gruppen einteilen. Einerseits gibt es CPU-lastigen Programme, die meist lange Berechnungen durchführen und dadurch eine hohe Laufzeit haben. Andererseits gibt es I/O-abhängige Programme, deren Laufzeit durch Input oder Output Aktionen bestimmt wird, beispielsweise eine Datenbankabfrage oder die Eingabe eines Benutzers. CPU-lastige Programme lassen sich durch paralleles Rechnen optimieren, also durch das Aufteilen größerer Rechnungen in kleinere Teilaufgaben. I/O-abhängige Programme lassen sich optimieren, indem man die andere Aufgaben erfüllen lassen kann, während das Programm auf ein I/O-Event wartet. Ich werde mich hier hauptsächlich mit CPU-lastigen Programmen beschäftigen, da diese für Hochleistungsrechnen sehr relevant sind. Unabhängige Threads werden auf unterschiedlichen Prozessoren ausgeführt. Jeder Prozessor besitzt ein eigenes Rechenwerk und ein eigenes Register, aber die Threads teilen sich den Speicher.

Paralleles Rechnen durch Multithreading ist in Python theoretisch möglich, praktisch gibt es aber keinen Gewinn an Performanz, da das Global Interpreter Lock von Python die Möglichkeiten des Multithreadings stark einschränkt. Darauf komme ich später noch zurück.

2 Multithreading in Python

2.1 API

Um Multithreading in Python nutzen zu können, muss lediglich die Thread Klasse aus dem threading Modul importiert werden.

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

Mit dem Konstruktor lässt sich ein neuer Thread starten. Mit target wählt man eine Zielmethode aus, die aufgerufen wird, sobald der Thread gestartet wurde. Mit args lassen sich die Parameter mitgeben, mit denen die target-Funktion ausgeführt werden soll. Nachdem der Thread initialisiert wurde, lassen sich unter anderem folgende Methoden aufrufen:

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

Mit der start()-Methode wird der Thread gestartet und die run() Methode wird aufgerufen. Die run()-Methode führt die über den target-Parameter mitgegebene Methode mit den übergebenen Parametern auf. Mit der join-Methode wird im Main-Thread auf das Beenden des Threads gewartet und wirft, falls die übergebene Timeout-Zeit überschritten wurde, eine Timeout-Exception. Die is_alive()-Methode gibt True zurück falls der Thread noch nicht seine Aufgabe beendet hat und gibt sonst False zurürck.

Damit kennen wir nun die wichtigsten Tools die man benötigt, um ein einfaches Programm durch Multithreading zu parallelisieren. Ich habe hier ein einfaches Referenzprogramm geschrieben, welches die Zeit misst, die benötigt wird um von 50.000.000 auf 0 herunter zu zählen.

```
1  import time
2
3  COUNT = 50000000
4
5  def countdown(n):
6      while n>0:
7          n -= 1
8
9  start = time.time()
10 countdown(COUNT)
11 end = time.time()
12
13 print('Time taken in seconds: ', end - start)
```

Dieses einfache Programm lässt sich wie folgt durch Threads parallelisieren:

```
1  import time
2  from threading import Thread
3
4  COUNT = 50000000
5  def countdown(n):
6      while n>0:
7          n -= 1
8
9  t1 = Thread(target=countdown, args=(COUNT//2,))
10 t2 = Thread(target=countdown, args=(COUNT//2,))
11 start = time.time()
12 t1.start()
13 t2.start()
14 t1.join()
15 t2.join()
16 end = time.time()
17 print('Time taken in seconds: ', end - start)
```

Das Ausführen beider Programme liefert folgendes Ergebnis: Auf einem Laptop mit 4 Intelcore i5-7300U Prozessoren mit 2.6 GHz Taktfrequenz braucht das erste Programm 4,47 Sekunden und das Zweite 8,19 Sekunden. Offensichtlich hat das Parallelisieren nicht den gewünschten Effekt bewirkt. Das liegt an Pythons Global Interpreter Lock, welches Multithreading zumindest in CPython verhindert. Aber um das GIL zu erklären, müssen wir uns zunächst die Probleme anschauen, die durch Multithreading hervorgerufen werden können.

3 Probleme und Gefahren des Multithreadings

Viele Operation, wie zum Beispiel das Inkrementieren eines Integerwertes, sind nicht atomar. Das bedeutet, dass mehr als eine Aktion auf Rechnerebene benötigt wird um solch eine Operation auszuführen. Für das Inkrementieren einer Zahl, muss diese Zahl zunächst aus dem Speicher geladen werden, anschließend inkrementiert werden, und schlussendlich zurück in den Speicher geschrieben werden. Beim Arbeiten mit mehreren Threads besteht nun die Gefahr, dass verschiedene Threads die selbe Variable verändern wollen und dadurch Informationen verloren gehen und überschrieben werden. Besonders problematisch ist dies für das Reference-Counting in der CPython Garbage Collection. Der Reference-Count könnte zu hoch sein und dadurch den Speicher mit unlöschbaren Objekten zumüllen oder Objekte löschen, obwohl noch Referenzen auf diese existieren.

Um dieses Problem zu bekämpfen können in vielen Programmiersprachen "Locks" benutzt werden. Diese können Objekte vor dem gleichzeitigem Zugriff von mehreren Threads schützen. In Python speziell sind Locks Teil des threading-Moduls. Diese Locks können 2 Zustände haben, "locked" und "unlocked". Die Methode acquire(blocking=True, timeout=-1) wartet darauf, dass das Lock den unlocked-Zustand erreicht und blockt bis dahin den Thread. Mit der Methode release() wird das Lock freigegeben. Wenn man alle Variablen, die von mehreren Threads verändert werden sollen, mit Locks schützt, verhindert man das Abfälschen von diesen Variablen. Allerdings kann ein neues Problem auftreten:

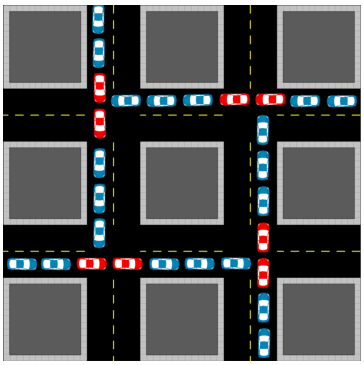


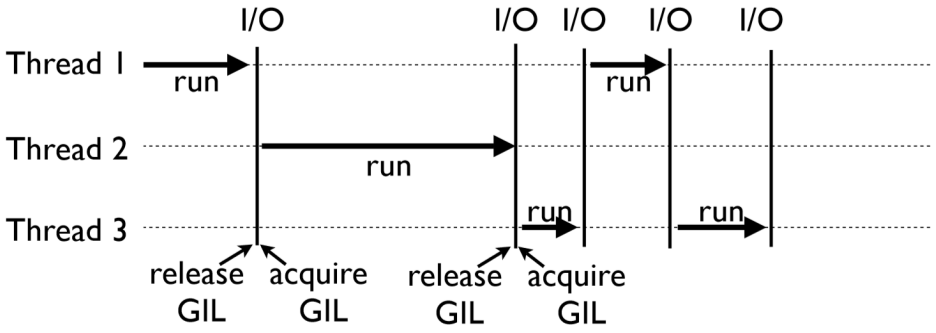
Figure 3.1: Deadlock: https://hackernoon.com/how-to-avoid-a-deadlock-while-writing-concurrent-programs-java-example-988bb07db25f

Deadlocks können entstehen, wenn unterschiedliche Threads Locks von anderen Threads benötigen und dadurch kommt das Programm zum Stillstand. Je mehr Locks benötigt werden, desto größer ist die Gefahr von Deadlocks. Deadlocks zu verhindern erfordert ein gutes Verständnis von Thread-Safety.

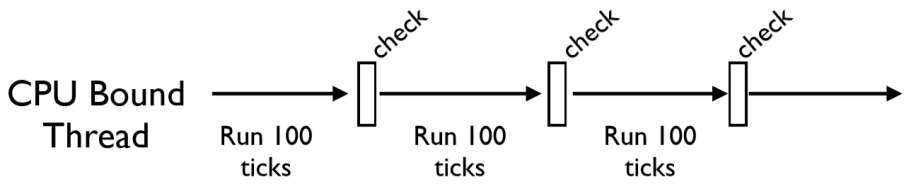
4 GIL

Um die Probleme von Thread-Safety zu beheben hat man sich recht früh in der Entwicklung von Python für ein Global Interpreter Lock entschieden. Das GIL belegt den Python-Interpreter mit einem einzigen Lock. Dadurch wird Thread-Safety für C-Extensions garantiert, also müssen sich Python-Entwickler nicht mit Thread-Safety beschäftigen um nützliche C-Extensions in ihren Programme zu benutzen. Außerdem bietet das GIL eine sehr hohe Performance für Programme mit nur einem einzigen Thread. Der große Nachteil ist, dass Multithreading, wie es in anderen Sprachen benutzt wird, in Python nicht möglich ist. I/O-Programme lassen sich trotz GIL immernoch durch Multithreading optimieren.

Das GIL unterscheidet zwischen I/O-Programmen und CPU-gebundenen Programmen. Bei I/O-Programmen wird das Lock nach jedem I/O Event freigegeben, also beispielsweise beim Lesen von Dateien, Schreiben, Datenbankzugriffen, etc.



Bei CPU-abhängigen Programmen wird das Lock nach 100 Ticks freigegeben. Dabei entspricht 1 Tick ungefähr einem Interpreter-Befehl. Die Anzahl der Ticks lässt durch sys.setcheckinterval() anpassen.



Eines der Probleme, das hierbei auftreten kann ist, dass die Threads um das GIL kämpfen und bestimmte Threads das GIL sofort wieder erhalten bevor andere Threads die Chance haben, das GIL zu nutzen. Darunter leidet die Performance von Programmen die mit Multithreading in Python arbeiten.

Warum hat man sich trotz offensichtlicher Nachteile für das GIL in der Entwicklung von Python entschieden und warum wurde das GIL in Python 3 nicht ersetzt durch eine Alternative, die Multithreading ermöglicht? Zunächst einmal war das GIL sehr leicht zu implementieren. Python ist außerdem eine sehr anfängerfreundliche Sprache. Python bietet einen Einstieg ins Programmieren, ohne großes Verständnis von Zeigern und Referenzen zu benötigen. Das GIL sorgt dafür, dass Python Entwickler sich nicht mit Thread-Safety beschäftigen müssen. Dadurch erlaubt es auch die einfache Benutztung von C/C++ Extensions. Dazu kommt, dass das GIL sehr hohe Performance für Programme mit nur einem Thread liefert.

Dazu sei gesagt, dass das GIL noch immer ein sehr umstrittenes Thema ist und viele Python-Entwickler wünschen sich eine Alternative, die auch Multithreading in Python erlaubt. Der "benevolent Diktator" von Python, Guido van Rossum, hat zu dem Thema folgendes Zitat geliefert:

„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."

Bisher gibt es noch keine Alternative für das GIL, die die Performance von Single-Threaded Programmen nicht verringert. Allerdings könnte man auf die CPython Implementation verzichten und beispielsweise mit Jython arbeiten. Dieses hat kein GIL, bringt allerdings auch andere Vor- und Nachteile gegenüber CPython mit sich.

Aber wir würden nicht über Python im Hochleistungsrechnen sprechen, wenn es keine Möglichkeit gäbe mit Python die Hardware eines Hochleistungsrechners effektiv zu nutzen. Als Alternative zum Multithreading gibt es das Multiprocessing.

5 Multiprocessing in Python

Der große Unterschied zwischen Threads und Prozessen liegt darin, dass Prozesse ihren eigenen Bereich im Speicher haben und einen eigenen Python-Interpreter besitzen. Somit hat jeder eigene Prozess auch ein eigenes GIL. Ein Nachteil von Prozessen ist allerdings, dass diese etwas CPU-lastiger sind als Threads und die Kommunikation zwischen Prozessen etwas komplizierter ist.

In Python funktioniert das Multiprocessing sehr ähnlich zum Multithreading. Aus dem multiprocessing muss die Prozess klasse importiert werden. Der Konstruktor sieht dem Thread Konstruktor sehr ähnlich.

- Process(target=None, name=None, args=(), kwargs={}, *, daemon=None)

Die wichtigsten Methoden der Prozess Klasse sehen denen aus der Thread Klasse ebenfalls sehr ähnlich und funktionieren alle analog.

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

Somit können wir nun unser anfangs geschriebenes Programm durch Multiprocessing optimieren:

```
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)
```

Der main-Block ist hier notwendig, da ansonsten die neu gestarteten Prozesse ebenfalls die main-Methode ausführen und dies zu einem Runtime-Error führt. Die erstellten Prozesse müssen, da sie sich nicht den Speicher des Main-Prozesses teilen, das Modul importieren.

Das Ausführen dieses Programms liefert auf dem selben Recher als Ausgabe 2,70 Sekunden. Dies entspricht nicht ganz der Hälfte der Zeit die für das Programm mit nur einem Prozessorkern, aber dies war zu erwarten, da das Erstellen der Prozesse ebenfalls etwas Zeit benötigt.

6 Zusammenfassung

Multithreading in Python ist zwar möglich, bietet aber nur bei I/O-Programmen einen Performance-Gewinn. Das GIL verhindert Multithreading für CPU-lastige Programme.

Multiprocessing funktioniert ähnlich zum Multithreading, umgeht allerdings das GIL mit einem eigenen Interpreter für jeden Prozess. Durch Multiprocessing lässt sich eine sehr hohe Performance für anspruchsvolle Rechnungen erreichen und ist besonders beim Hochleistungsrechnen sehr wichtig.

7 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