Duale Hochschule Baden-Württemberg
Mannheim

Hausarbeit

Multiprocessing mit Python

Studiengang Wirtschaftsinformatik

Studienrichtung Data Science

Verfasser/in:	Lukas Benner, Hanna Siegmann
Firma:	Würth IT GmbH
Kurs:	WWI19DSC
Vorlesung:	Fortgeschrittene Programmierung
Dozent:	Dr Zoufiné Lauer-Baré zoufine.bare@gmail.com
Bearbeitungszeitraum:	04.06.2020 – 03.07.2020

Inhaltsverzeichnis

1 Theoretischer Hintergrund

Um die Laufzeit eines Pythons Skript zu verbessern, ist eine Möglichkeit das Modul Multiprocessing einzusetzen. Dieses Modul hat als Ziel die parallele Programmierung zu ermöglichen und somit Teile des Codes gleichzeitig ausführbar zu machen. In der folgenden Hausarbeit wird aufgezeigt, wie man mit der Implementierung dieses Modul Zeit sparen kann und welche Vorteile es gegenüber einer einfachen sequenziellen Programmierung bietet.

Für die Programmiersprache Python gibt es neben Multiprocessing viele verschiedene Module, die die Parallelisierung in der Programmierung unterstützen. Weiter wichtige Module sind „threading“ und „parallel“. Multiprocessing ist das jüngste Modul und unterstützt im Gegensatz zu den anderen Modulen Prozessoren mit mehreren Kernen oder Multiprozessorsystemen entscheidend besser. Aus diesem Grund hat das Modul heutzutage mehr Relevanz.

Die grundlegende Funktionsweise um mehrere Prozesse gleichzeitig auszuführen, beruht darauf, dass Teilaufgaben eines Prozesses verteilt werden. Jeder Kern der CPU (bzw. virtueller Kern) bekommt eine Teilaufgabe. Dies ist möglich, da die Kerne größtenteils getrennt voneinander rechnen können. Alle Teilaufgaben werden nun gleichzeitig und unabhängig voneinander ausgeführt. Das Programm startet dazu mehrere Exemplare von sich selbst, um diese Teilaufgaben auszuführen.[1]

Grund dafür ist die Konzeption der Sprache Python. Um ein Speicherverwaltungsproblem zu umgehen wurde die GIL (global interpreter lock) eingeführt, welche dem Interpreter nur erlaubt ein Thread auszuführen. Daraus folgt, dass nur eine CPU verwendet werden kann. Um dies zu umgehen, wird das gleiche Skript mehrmals aufgerufen und umgeht somit die GIL.[2] Dieser Ablauf wird als "Forken"bezeichnet.

Durch das Forken können theoretisch mehr Rechnungen gleichzeitig ausgeführt werden, dadurch kommt es zu einem zeitlichen Vorteil gegenüber der sequenziellen Programmierung. Dieser ist zunächst einmal kritisch zu betrachten, denn sowohl die Kommunikation und Synchronisierung während der Parallelisierung, sowie das Erzeugen von neuen Prozessen durch das Betriebssystem verbraucht Ressourcen. Insbesondere bei kleinen Prozessen ist durch die Implementierung von Multiprocessing kein signifikanter Unterschied in der Ausführungszeit zu beobachten oder es findet eine Verlangsamung statt. Daraus schließt sich, dass eine bestimmte Größe des Prozesses vorhanden sein muss, damit sich Multiprocessing zeitlich lohnt.[3]

[1] Vgl. Kaminski 2016, S.290.
[2] Vgl. Urban Instiute (Data@Urban) 2018.
[3] Vgl. Kaminski 2016, S.290.

2 Das Module Multiprocessing

Das Module Multiprocessing zählt zu den Standardbibilotheken in Python. Es umfasst zwei wichtige Klassen. Zum einen die Klasse Process und zum anderen die Klasse Pool. Der Unterschied beider Klassen besteht darin, dass Processes jede einzelne Teilaufgabe separat zu einem Prozessor (Kern) schickt und Pool Pakete von Teilaufgaben schickt. Pool zu verwenden lohnt sich dann, wenn man viele kleine Teilaufgaben hat, die jeweils schnell berechnet werden können. Process hingegen ist in der Verwendung sinnvoll, wenn es nur wenig aber rechenintensive Teilaufgaben gibt.[4]

Für die spätere Implementierung am eigenen Beispiel ist die Klasse Pool sinnvoller, deshalb wird sich im Folgenden auf die Erklärung von Pool beschränkt. Dazu wird die Funktionsweise am nachfolgenden Codebeispiel[5] beschrieben.

```
1 from multiprocessing import Pool
2 def f(x):
3     return x*x
4 if __name__ == '__main__':
5     with Pool(3) as p:
6         result = (p.map(f, [1, 2, 3, 4]))
```

Zuerst wird der benötigte Befehl vom Modul Multiprocessing importiert (Zeile 1). Anschließend wird die zu parallelisierende Funktion definiert (Zeile 2). Danach ist eine Kontrollstruktur aufgebaut, um zu gewährleisten, dass bei der Parallelisierung die entstehenden Instanzen nicht erneut den Pool-Befehl ausführen, sondern nur die Möglichkeit haben auf die Funktion f zuzugreifen. Der Befehl Pool(3) erzeugt 3 parallele Instanzen. Dies bedeutet, dass das Script gleichzeitig 3 Mal erzeugt wird. Diese Instanzen führen gleichzeitig die Berechnung der Funktion f durch und können aufgrund der wie oben erklärten IF-Verzweigung kein neuen Pool erzeugen, da diese Instanzen nicht den Namen „main“ tragen. Die Ergebnisse aller Instanzen laufen nun in der Variable „result“ zusammen.

[4] Vgl. Urban Instiute (Data@Urban) 2018.
[5] Vgl. *multiprocessing — Process-based parallelism — Python 3.8.3 documentation* 2020.

3 Implementierung des Multiprocessing-Scripts

Um den zeitlichen Vorteil von Multiprocessing aufzuzeigen, wird hier eine Funktion verwendet, welche jeden Abstand von einem Punkt zu einer Liste mit Punkten herausfindet. Diese Liste wird sortiert und könnte jetzt zum Beispiel verwendet werden, um die nächsten Punkte ausfindig zu machen. (Dies wäre das Grundprinzip des KNN Algorithmus).

Im Skript werden zu Beginn alle relevanten Bibliotheken geladen. Anschließend gibt es eine Funktion (create_points), welche unter Angabe des Parameters der Listenlänge, eine Liste mit zufälligen dreidimensionalen Punkten erzeugt. Danach werden diese Punkte und ein fester x1 Punkt zur Berechnung des Abstands an eine Funktion mit und ohne Multiprocessing gegeben.

3.1 Beispiel ohne Multiprocessing

Die Funktion für die Berechnung des Abstands ist folgendermaßen aufgebaut:

```
1  def get_distances_classic(x1, points):
2      'calculate␣distance␣to␣every␣other␣point␣without␣mutliprocessing'
3      distances = []
4      t0 = time.time()
5      for p in points:
6          distance = calc_distance(x1,p)
7          distances.append(distance)
8      distances.sort()
9      print(f'Zeit␣(singleprocessing):␣{(time.time()-t0):.8f}s')
10     return distances
```

Zuerst wird eine leere Liste erstellt (Zeile 3), in welcher später die Abstände gespeichert werden. Der Timer in Zeile 4 und 9 misst die benötigte Zeit für die Rechenschritte, um sie später mit der Funktion mit Multiprocessing vergleichen zu können. Dazu wird die Bibliothek „time“ verwendet.

Im nächsten Schritt wird mithilfe der for-Schleife über die Liste der zufällig erstellten Punkte iteriert. Während jeder Iteration wird der Abstand von einem Punkt aus der Liste zu dem Punkt x1 berechnet. Dazu wird eine andere Funktion aufgerufen, diese

haben wir zuvor im Code definiert und sie verwendet den einfachen euklidischen Abstand. Der berechnete Abstand wird nun zu der Liste „distance“ hinzugefügt. Diese wird nach der for-Schleife sortiert und von der Funktion zurückgegeben.

3.2 Beispiel mit Multiprocessing

Der Code mit Multiprocessing ähnelt dem vorherigen Code, der einzige Unterschied liegt darin, dass anstatt der for-Schleife eine if-Kontrollstruktur verwendet wird.

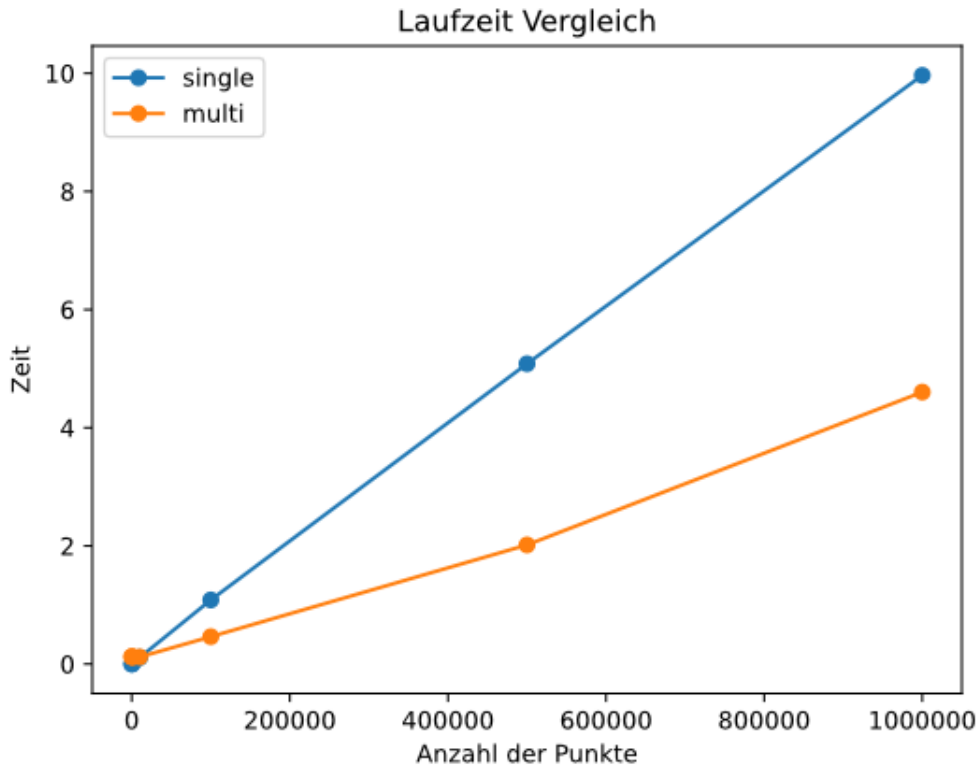
```
1  def get_distances_multi(x1, points):
2      'calculate␣distance␣to␣every␣other␣point␣with␣mutliprocessing'
3      print("--Berechne␣Abstand␣(multi)--")
4      distances = []
5      t0 = time.time()
6      if __name__ == '__main__':
7          with Pool(10) as p:
8              distance = p.starmap(calc_distance,[(x1,p) for p in points])
9              distances.append(distance)
10     distances.sort()
11     print(f'Zeit␣(multiprocessing):␣{(time.time()-t0):.8f}s')
12     return distances
```

Diese if-Kontrollstruktur ist, wie in Kapitel 2 erklärt, für die Anwendung von Pool wichtig. In diesem Codebeispiel werden 10 Pools verwendet. Die Aufgabe der Berechnung wird also in 10 Teilaufgaben geteilt und parallel berechnet. Sie läuft in der Variable „distance“ zusammen.

4 Fazit und Rückschlüsse

Vergleicht man abschließend die zeitliche Ausführung beider Funktionen, kann folgendes festgestellt werden:

In diesem Schaubild ist zu erkennen, dass sich in diesem Fall Multiprocessing schon ab sehr wenigen Iterationen (Anzahl der Punkte) lohnt, auch wenn dieser sehr gering ist und es sich hierbei um einen Idealfall handelt.



Wichtig festzuhalten ist, dass an jedem Endgerät oder Server die Laufzeiten anders ausfallen können. Abhängig ist dies von der verwendeten Hardware und der Systemauslastung durch andere Prozesse.

Zusammenfassend kann festgehalten werden, Multiprocessing kann sich unter bestimmten Umständen lohnen. Ein Kriterium dafür ist, dass der Prozess groß genug bzw. lang genug dauert. Ansonsten ist der zusätzliche Rechenaufwand zum Aufteilen und Zusammenführen relativ gesehen zu groß. Außerdem muss die Funktion unabhängig von anderen Instanzen sein, sodass eine Parallelisierung überhaupt technisch möglich ist.

Literaturverzeichnis

Kaminski, Steffan (2016). *Python 3*. De Gruyter Studium. Berlin ; Boston: De Gruyter Oldenbourg. 482 S. ISBN: 978-3-11-047361-2.

multiprocessing — Process-based parallelism — Python 3.8.3 documentation (2020). URL: `https://docs.python.org/3/library/multiprocessing.html` (besucht am 04. 06. 2020).

Urban Instiute (Data@Urban) (18. Sep. 2018). *Using Multiprocessing to Make Python Code Faster*. URL: `https://medium.com/@urban_institute/using-multiprocessing-to-make-python-code-faster-23ea5ef996ba` (besucht am 10. 06. 2020).

Ehrenwörtliche Erklärung

Wir versichern hiermit, dass wir die vorliegende Arbeit mit dem Thema: *Multiprocessing mit Python* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Ort, Datum Lukas Benner, Hanna Siegmann