

LAUFZEITOPTIMIERUNG

Ausführungsgeschwindigkeit von Python-Programmen Messen und Verbessern

Folien: C. Statz, S. Voigt, C. Knoll
Sprecher: S. Brückner

Dresden, WiSe 2013/14

- Einführung
- Zeitmessung
- Allgemeine Tipps
- Beschleunigung durch C-Extensions

Was ist Performance?

- Laufzeit
 - Speicherbedarf (RAM, Festplatte)
 - Energie
- Hier: nur Laufzeit (einfach messbar, oft am wichtigsten)

Fakten

- Python ist langsamer als kompilierte Sprachen (Interpreter)
 - Oft aber nicht störend/wahrnehmbar (0.1s vs. 0.01s)
 - Laufzeitorientierung von Code ist selbst oft sehr zeitaufwändig
- Zielkonflikt Ausführungs- vs. Entwicklungszeit

Was ist Performance?

- Laufzeit
 - Speicherbedarf (RAM, Festplatte)
 - Energie
- Hier: nur Laufzeit (einfach messbar, oft am wichtigsten)

Fakten

- Python ist langsamer als kompilierte Sprachen (Interpreter)
 - Oft aber nicht störend/wahrnehmbar (0.1s vs. 0.01s)
 - Laufzeitoroptimierung von Code ist selbst oft sehr zeitaufwändig
- Zielkonflikt Ausführungs- vs. Entwicklungszeit
-
- ⇒ Allgemeine Tipps von Anfang an beachten
- ⇒ Spezifische Optimierung laufzeitkritischer Algorithmen-Teile

- Modul `time`
- `time.time()` gibt Epoch-Zeit in Sekunden (auch UNIX-Time, die vergangenen Sekunden seit dem 01.01.1970)
- Vorteil: sehr einfach
- Nachteil: zusätzlicher („boilerplate“)-Code im Programm verteilt

```
import time
```

```
s = 0
```

```
start = time.time()
```

```
for i in range(100000):  
    s += i**(0.5)
```

```
print "Dauer [s]:", time.time() - start
```

- Modul `timeit`
- Laufzeitmessung eines Statements (meist Funktionsaufruf)
- Gut für Vergleich von Codesnippets für spezielles Problem
- Statement muss als string oder callable übergeben werden
- Vorteile: nicht invasiv, Mittelung mehrerer Durchläufe

```
import timeit
import math
```

```
def wurzel_v1():
    return math.sqrt(2)
```

```
def wurzel_v2():
    return 2**0.5
```

```
print timeit.timeit("2**0.5", number=100000)
print timeit.timeit(wurzel_v1, number=100000)
print timeit.timeit(wurzel_v2, number=100000)
```

- Modul cProfile: detaillierte Laufzeitanalyse eines (ggf. sehr umfangreichen) Programms → Flaschenhals finden
- Profiling erzeugt Overhead – Programm läuft etwas langsamer ab als ohne
- Ergebnisse als print-Ausgabe oder in Datei zur Nutzung in Analysewerkzeugen
- Argument wird als String übergeben, cProfile auch in der Kommandozeile nutzbar

```
import cProfile
```

```
def main():  
    s = 0
```

```
    for i in range(1000000):  
        s += math.sqrt(i)
```

```
cProfile.run("main()")
```

Alternativ: Kommandozeilenaufruf:

```
python -m cProfile -o test.prof test.py
```

Ausgabe des Beispiels:

100004 function calls in 0.036 seconds

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.022	0.022	0.036	0.036	<module1>:34(main)
1	0.000	0.000	0.036	0.036	<string>:1(<module>)
100000	0.013	0.000	0.013	0.000	{math.sqrt}
1	0.000	0.000	0.000	0.000	{method [...]}
1	0.001	0.001	0.001	0.001	{range}

- Zeigt: welche Funktion wie oft aufgerufen wurde und wieviel Zeit dafür benötigt wurde
- Ansatzpunkte für Optimierung finden
- Hier interessant: nur ca. 1/3 der Laufzeit für sqrt benötigt
 - Rest ist Overhead (Funktion, Schleife)

Ausgabe des Beispiels:

100004 function calls in 0.036 seconds

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.022	0.022	0.036	0.036	<module1>:34(main)
1	0.000	0.000	0.036	0.036	<string>:1(<module>)
100000	0.013	0.000	0.013	0.000	{math.sqrt}
1	0.000	0.000	0.000	0.000	{method [...]}
1	0.001	0.001	0.001	0.001	{range}

- Zeigt: welche Funktion wie oft aufgerufen wurde und wieviel Zeit dafür benötigt wurde
- Ansatzpunkte für Optimierung finden
- Hier interessant: nur ca. 1/3 der Laufzeit für sqrt benötigt
 - Rest ist Overhead (Funktion, Schleife)
 - Weitergehende Analyse: Modul pstats

- Code erst optimieren, wenn tatsächlich Bedarf besteht:
 - „Premature optimisation is the root of all evil.“
- Profiling nutzen und nur die lohnenswerten Stellen zu identifizieren
- Nur korrekten Code optimieren
 - Unit-Tests nutzen, um Richtigkeit des Codes während der Überarbeitung sicherzustellen
 - Reihenfolge: „Make it work. Make it right. Make it fast.“
 - Spezielle Bibliotheken für jeweiliges Problem verwenden
 - Z. B. numpy für Numerik
 - Selbst in C/Fortran geschrieben → viel schneller als reines Python

- Angemessene Datentypen verwenden: `tuple` oder `dict` statt `list`
- Beispiel: „Element Lookup“

```
3 in {1:1, 2:2, 3:3} # Aufwand  $O(1)$   
3 in [1, 2, 3] # Aufwand  $O(n)$ 
```

- „Punkte“ (Objektorientierung) vermeiden:
 - jeder Punkt bedeutet Attribute/Member Lookup,
 - lokales Zwischenspeichern lohnt sich besonders in Schleifen

```
wurzel = math.sqrt  
wurzel(2)
```

- C-Unterbau nutzen, z.B. `map()` statt `for`-Schleife

```
map(str, [1, 2, 3])  
# statt  
r = []  
for i in [1, 2, 3]:  
    r.append(str(i))
```

- Funktionalität möglichst „von innen nach außen“ verlagern
 - import-Statements
 - Initialisierungen von Variablen
 - Berechnungen → Zwischenergebnisse Speichern/Cachen
- Iteratoren nutzen (z.B. `xrange`) statt `range`): erzeugen Funktion, um nächstes Element zu berechnen, sonst wird komplette Datenstruktur für die Iteration im Vorfeld generiert
- Lokale Variablen verwenden
 - der Zugriff ist hier schneller, als auf Variable außerhalb des aktuellen Namensraums
 - Funktionen vektorisieren für schnelle Array-Operationen (`numpy.vectorize`)

Es gibt mehrere Möglichkeiten:

- „Just in Time“-Kompilierung von bestimmte Code abschnitten (Modul weave)
- Übersetzen des Python Codes in cython (sehr ähnlich)
- ctypes
 - Kann (selbst geschriebene) externe Bibliotheken (z.B. *.dll unter Windows) in Python laden
 - sehr mächtig und flexibel
 - Hier nicht näher betrachtet

- Laplace-Gleichung (z.B. 2D-Wärmeleitung): $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$
- Gesucht $u(x, y)$ (Temperaturverteilung)
- Iteratives numerisches Lösen: → sehr viele (simple) Array-Zugriffe
- Anwendung von `scipy.weave`:

Umschreiben des Numpy-Ausdrucks als String:

```
expr = "u[1:-1,1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2"+  
      " + (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv"
```

String an `weave.blitz` uebergeben:

```
weave.blitz(expr, check_size=0)
```

- Aus dem Numpy-Ausdruck wird C-Code erstellt
- Der Code wird automatisch bei Aufruf kompiliert und eingebunden

- Cython ist eigene Programmiersprache
 - Sehr eng an Python angelehnt aber mit expliziten statischen Typ-Informationen
- Kann in C übersetzt werden → kompilierbar → schneller
- Vorgehen:
 - Algorithmus in Python Entwickeln
 - Python manuell nach Cython übersetzen
 - Cython-Code nach C übersetzen lassen
 - C-Code kompilieren
 - So erstelltes Modul „ganz normal“ importieren

Wieder Laplace-Gleichung

```
# Cython Datei
# Abschalten der Ueberpruefung der Eingabedaten
#cython: boundscheck=False
#cython: wraparound=False

# Importieren der C-Header von numpy
cimport numpy as np

# Definieren der Eigentlichen iterativen Update-Funktion
def update(np.ndarray[double, ndim=2] u, double dx2, double dy2):
    cdef unsigned int i, j
    for i in xrange(1, u.shape[0]-1):
        for j in xrange(1, u.shape[1]-1):
            u[i,j] = ((u[i+1, j] + u[i-1, j])*dy2 +
                      (u[i, j+1] + u[i, j-1])*dx2) / (2*(dx2+dy2))
```