A Crash Course in Python Performance

From identifying slow code, to calling c libraries from your python.

by Tommi Kabelitz (University of Adelaide)
on 25/03/2022

- git@github.com:TommiKabelitz/FastPython.git
- https://github.com/TommiKabelitz/FastPython

Profiling

» Identifying slow code

INVEST YOUR TIME WHERE YOUR CODE TAKES TIME

* You may think you know what is slow - You will often be wrong
* Profilers track the calls to functions, keeping track of how long is spent where
* Reports are (normally) easy to read and sort

» Identifying slow code

* Note: slight overhead associated with using profilers
* Use at the start to work out which functions are slow
* When improving functions, use modules like timeit to simply time the function without overhead
* I recommend cProfile for python

» Using the profiler

Module imports

```
#The profiler
import pstats, cProfile

#For tidying the output
import io
from pstats import SortKey
```

» Using the profiler

Running the profiler

```
#Initialising the profiler
prof = cProfile.Profile()

#Profiling some function
prof.enable()
function_to_profile()
prof.disable()
```

» Using the profiler

Outputting the stats

```
#A little messing around is required to format the output
s = io.StringIO()
sortby = SortKey.CUMULATIVE #Alternatively TIME, CALLS, etc
ps = pstats.Stats(prof, stream=s).sort_stats(sortby)
ps.print_stats(100)
#Now the output is in s, so either
print(s.getvalue())
#or
with open('profile.txt','w') as f:
        f.write(s.getvalue())
```

» Using the profiler

Profiler output (Example from my plotting code)

```
        42441171 function calls (40411093 primitive calls) in 30.739 seconds

   Ordered by: cumulative time
   List reduced from 2678 to 100 due to restriction <100>

   ncalls      tottime  percall  cumtime  percall filename:lineno(function)
        1        0.000    0.000   30.767   30.767  effectiveMass.py:21(main)
        1        0.018    0.018   30.750   30.750  effectiveMass.py:68(DoCombination)
      254        0.001    0.000   24.578    0.097  /*condapath*/matplotlib/backends/backend_pdf.py:2464(savefig)
      254        0.001    0.000   24.545    0.097  /*condapath*/matplotlib/figure.py:2063(savefig)
      254        0.005    0.000   24.543    0.097  /*condapath*/matplotlib/backend_bases.py:2001(print_figure)
      254        0.003    0.000   24.444    0.096  /*condapath*/matplotlib/backends/backend_pdf.py:2532(print_pdf)
35808/254        0.088    0.000   24.352    0.096  /*condapath*/matplotlib/artist.py:30(draw_wrapper)
      254        0.006    0.000   24.351    0.096  /*condapath*/matplotlib/figure.py:1688(draw)
      256        0.001    0.000   12.555    0.049  /*condapath*/matplotlib/cbook/deprecation.py:347(wrapper)
      256        0.004    0.000   12.547    0.049  /*condapath*/matplotlib/figure.py:2448(tight_layout)
      256        0.004    0.000   12.430    0.049  /*condapath*/matplotlib/tight_layout.py:264(get_tight_layout_fig
ure)
      256        0.010    0.000   12.384    0.048  /*condapath*/matplotlib/tight_layout.py:33(auto_adjust_subplotpa
rs)
      256        0.001    0.000   12.310    0.048  /*condapath*/matplotlib/tight_layout.py:109(<listcomp>)
      256        0.037    0.000   12.309    0.048  /*condapath*/matplotlib/axes/_base.py:4270(get_tightbbox)
  508/254        0.007    0.000   11.743    0.046  /*condapath*/matplotlib/image.py:119(_draw_list_compositing_imag
es)
```

» Using the profiler

Columns

* ncalls: Number of function calls. Denominator of fraction (if present) denotes number of recursive calls
* tottime: Time spent in a function, excluding time spent in other functions
* percall: tottime/ncalls
* cumtime: Total time spent in a function, including calls to subfunctions. Accurate for recursive functions
* percall: cumtime/ncalls
* filename:lineno(function) The function to which the stats refer

» Using the profiler

Takeaways about my plotting code

* Majority of time is spent saving figures to file
* Time reading/manipulating data is minimal
* If I want to speed up this code (I don't, not worth it)
    * Save speed is only important factor. Ideas:
    * Plots per page
    * Maybe pdf is not ideal
    * Different package?
    * etc.

» Profiling recap

* When you care about speed, profile. It is easy
* You will be surprised at which piece of code is the slow part
* You invest your time improving the code that is slow

Weaknesses

» A word on why python can be slow

We cannot write efficient python without knowing what makes it slow. Hence, weaknesses:

* Interpreted not compiled
* Duck typing
* The GIL (Global Interpreter Lock)

» A word on why python can be slow

A simple example

```
n = 10
scale = 0.4
if scale > 1:
        total = 0
        for i in range(1,n+1):
                total += scale*i
else:
        total = 1
        for i in range(1,n+1):
                total *= (scale - i)
```

Leverage other languages

» Integrating other languages

* Why python? $\rightarrow$ easy to write
* But we tradeoff speed for that
* Consider mixing languages

» Integrating other languages

- Do the heavy computation in c, c++, fortran, etc.
- Do the fiddly setup in python
  - File paths
  - String manipulation
  - Data cleaning
  - Parameter organisation
  - etc.

» Integrating other languages

The subprocess module is excellent for running other code. A simple example:

```
#the command to run, each argument as an element
executable = ['path/to/executable.x']
report_file = 'path/to/reportfile.txt'
input_file = 'path/to/input/file.txt'

generate_input_file(input_file) #Generate your input file(s)

with open(report_file,'w') as f:
        subprocess.run(command,
                       input=input_file + '\n',
                       text=True,
                       stdout=f,
                       stderr=subprocess.STDOUT,
                       timeout=600) #timeout in seconds

# Note: don't just copy paste this entire code block,
# the indentation is broken because Tex hates me.
```

Vectorisation

» Writing *fast* pure-python

* Cannot expect speed in any language if the code is *bad*
* Vectorisation is vital
* Many python libraries provide highly vectorised routines already

Vectorisation

The transformation of code to act on entire arrays at once, rather than element by element

» Writing *fast* pure-python

Vectorisation steps

* Remove loops where possible
* Replace loops with vectorised operations

```
import numpy as np
n = 10
scale = 0.4
array = np.arange(1,n+1)
if scale > 1:
        total = scale*sum(array)
else
        total = np.prod(scale - array)
```

» Writing *fast* pure-python

* numpy, scipy and libraries written on those are likely to be highly vectorised
* So use functions from there as often as possible
* There are a lot
* But sometimes creativity is required

MPI

» Leveraging multiple cores

* Very possible in python
* Several libraries for it
* Definitely something to consider when you do not need any message passing
* Still possible with message passing, but more painful

DISCLAIMER: I haven't ever used multiple CPUs in python as I haven't needed to. I just want to mention it.

Cython

* A basic intro
  * For directly integrating other languages
  * For turning python code into c code

» What is cython

* Compiles python into c (or c++)
* Functions can be directly imported into python
* Can import c and fortran functions too
* Basics are very simple, plenty of complex stuff too

» Installation

Installation:

```
python -m pip install cython
```

* File extension is .pyx
* Compilation will produce a .so file
* Then simply import as normal

» Compilation

Compile using

```
python setup.py build_ext --inplace
```

Where the setup.py file is

```
from setuptools import setup
from Cython.Build import cythonize

setup(ext_modules=cythonize('cython_module.pyx'),
          compiler_directives={'language_level":3},
          zip_safe=False)
```

» The .pyx file

In terms of preparing the .pyx file all you need is:

```
import cython

# Optional compiler directives
# cython: boundscheck=False, wraparound=False
# cython: initializedcheck=False
```

Cython

* A basic intro
  * For directly integrating other languages
  * For turning python code into c code

» Definitions

The most important pieces of cython syntax:

* cdef - A purely c, fastest
* cpdef - A hybrid object, slower, importable in python

Rule of thumb, if you don't need to use/access the value from within your python. Use cdef.
Only cpdef the functions/objects you need at the end

» Typing

You do not have to type everything. The more you type, the faster your code.

```
#Function definition (returns a 128bit int)
cpdef long my_fun(int a, float b,
                  str s, double[:] array):

    cdef int i,j,k
    cdef double* array_pointer = &array[0]
    cdef int size = array.shape[0]
    c = 2*a    #Note not typed, but could be
```

Cythonth

* A basic intro
  * For directly integrating other languages
  * For turning python code into c code

» Cython for integrating other languages

* subprocess.run is not bad
* Still awkward passing information between languages
* Cython allows direct calling of c, c++ and fortran subroutines
* And it actually is pretty simple

» Cython for integrating other languages

A little involved for slides, so see the github repo in cython/interfacing/fortran. Files required: (and the order to look at them)

* fortmod.f90 (Fortran module containing code to import)
* fort_interface.f90 (Interfacing the fortran to c)
* fort_interface.h (c header file)
* fort_interface.pyx (cython file which interfaces the python and fortran)
* setup.py (Has to change to link the code properly)
* runscript.py (Does the actual call)

Cython

* A basic intro
  * For directly integrating other languages
  * For turning python code into c code

» What python should I compile with cython?

Compiling all of your python is a waste of time. Have to recompile every time, cython only helps with computation speed.

Compile:

* The functions that take lots of time in your profile report
* (Only if typing is the bottleneck)
* Functions that get called often
* Functions which cannot be vectorised easily
* Functions which you don't touch often

Examples:

* Jackknife/Bootstrap routines
* Functions that are passed to optimisation routines

» Abrubt ending

Takeaways

* Please profile your code
* Various options for speedup (vectorisation, mpi, interfacing other languages)
* Cython is also an option
* Either use it to type your code
* IMO its true calling is direct interfacing

Questions?