Distributed Computing and Introduction to High Performance Computing

Imad Kissami[1], Nouredine Ouhaddou[1]
[1]Mohammed VI Polytechnic University, Benguerir, Morocco

Outline of this lecture

- About Python
- Python is slow
- Profiling a Python code

About Python

- Python was created by Guido van Rossum in 1991 (last version 3.9 - 05/10/2020)
- Python is **simple**
- Python is **fully featured**
- Python is **readable**
- Python is **extensible**
- Python is **ubiquitous, portable, and free**
- Python has **many third party libraries, tools, and a large community**

About Python

- Python was created by Guido van Rossum in 1991 (last version 3.9 - 05/10/2020)
- Python is **simple**
- Python is **fully featured**
- Python is **readable**
- Python is **extensible**
- Python is **ubiquitous, portable, and free**
- Python has **many third party libraries, tools, and a large community**

➠ **But Python is slow!!**

About Python

- Python was created by Guido van Rossum in 1991 (last version 3.9 - 05/10/2020)
- Python is **simple**
- Python is **fully featured**
- Python is **readable**
- Python is **extensible**
- Python is **ubiquitous, portable, and free**
- Python has **many third party libraries, tools, and a large community**

➠ **Does is really matters?**

Python is slow

When does it matter?

- Is my code fast enough to produce the results I need in the time I have?
- How many CPUh is this code going to waste over its lifetime?
  - How inefficient is it?
  - How long does it run?
  - How often will it run?
- Does it cause problems on the system it's running on?
- How much effort is it to make it run faster?

➠ For those who are interested, you can follow this MOOC

Profiling a Python code

Why?

- It's mandatory to know what sections of code are bottlenecks in order to improve performance.
- You need to measure it, not to guess it
- **Premature optimization is the root of all evil** D. Knuth
- **First make it work. Then make it right. Then make it fast**. K. Beck
- How?

Profiling a Python code

Different kinds of profilers

- Deterministic and statistical profiling
  - the profiler will be monitoring all the events
  - it will sample after time intervals to collect that information
- The level at which resources are measured; module, function or line level
- Profile visualizers

Profiling a Python code

Available tools

- **Inbuilt timing modules**
- **profile and cProfile**
- pstats
- **line_profiler**
- Yappi
- vmprof-python
- pyinstrument
- gprof2dot
- pyprof2calltree, KCacheGrind
- **snakeviz**
- Scalene

Profiling a Python code

Use case

```
1  def linspace(start, stop, n):
2    step = float(stop - start) / (n - 1)
3    return [start + i * step for i in range(n)]
4
5  def mandel(c, maxiter):
6    z = c
7    for n in range(maxiter):
8      if abs(z) ¿ 2:
9        return n
10     z = z*z + c
11   return n
12
13 def mandel_set(xmin=-2.0, xmax=0.5, ymin=-1.25, ymax=1.25,
14                 width=1000, height=1000, maxiter=80):
15   r = linspace(xmin, xmax, width)
16   i = linspace(ymin, ymax, height)
17   n = [[0]*width for _ in range(height)]
18   for x in range(width):
19     for y in range(height):
20       n[y][x] = mandel(complex(r[x], i[y]), maxiter)
21   return n
```

Profiling a Python code

timeit

The very naive way

```
1  import time
2
3  start_time = time.time()
4  mandel_set()
5  end_time = time.time()
6  # Time taken in seconds
7  elapsed_time = end_time - start_time
8
9  print('> Elapsed time', elapsed_time)
```

or using the magic method `timeit`

```
1    [In] %timeit mandel_set()
2    [Out] 3.01 s +/- 84.6 ms per loop (mean +/- std. dev. of 7 runs, 1 loop each)
```

Profiling a Python code

prun

```
[In] %prun -s cumulative mandel_set()
```

which is, in console mode, equivalent to

```
python -m cProfile -s cumulative mandel.py
```

```
         25214601 function calls in 5.151 seconds

   Ordered by: cumulative time

   ncalls tottime percall cumtime percall filename:lineno(function)
        1 0.000 0.000 5.151 5.151 {built-in method builtins.exec}
        1 0.002 0.002 5.151 5.151 <string>:1(<module>)
        1 0.291 0.291 5.149 5.149 <ipython-input-4-9421bc2016cb>:13(mandel_set)
  1000000 3.461 0.000 4.849 0.000 <ipython-input-4-9421bc2016cb>:5(mandel)
 24214592 1.388 0.000 1.388 0.000 {built-in method builtins.abs}
        1 0.008 0.008 0.008 0.008 <ipython-input-4-9421bc2016cb>:17(<listcomp>)
        2 0.000 0.000 0.000 0.000 <ipython-input-4-9421bc2016cb>:1(linspace)
        2 0.000 0.000 0.000 0.000 <ipython-input-4-9421bc2016cb>:3(<listcomp>)
        1 0.000 0.000 0.000 0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

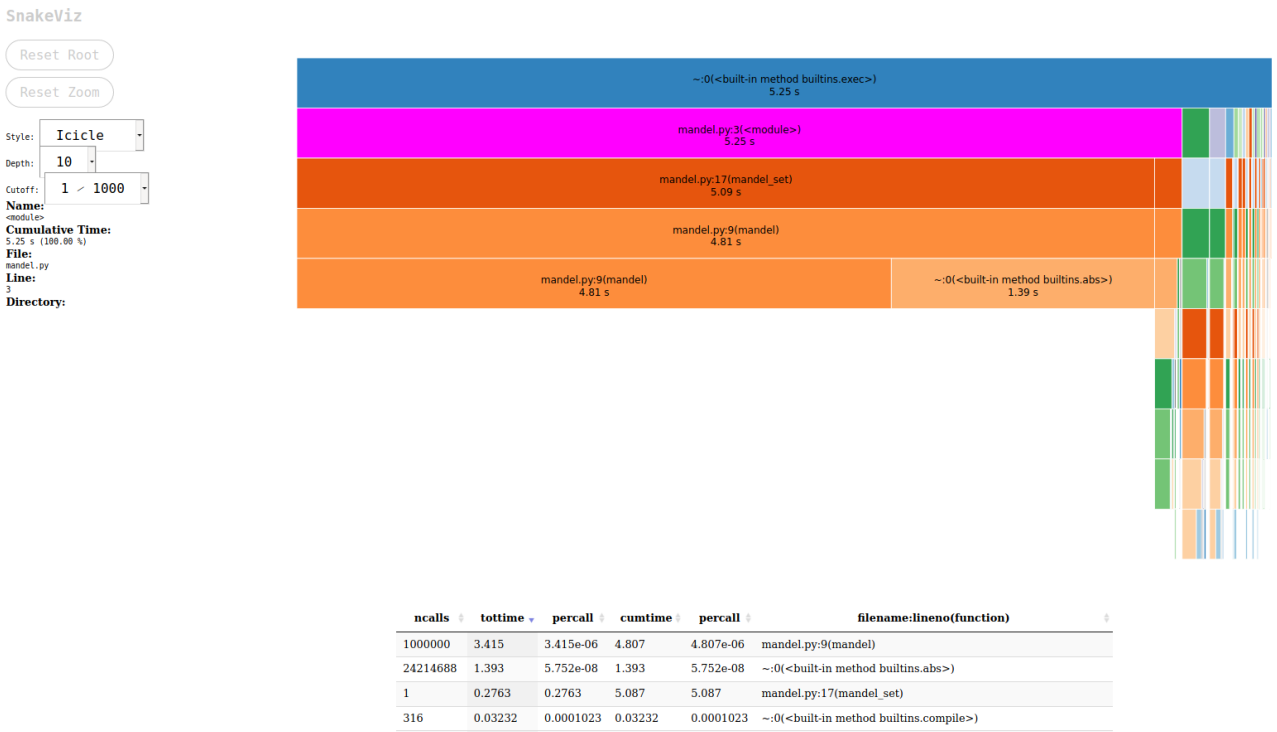
- Most of the time is spent in the `mandel` function
- profiling introduces some overhead 5.14(s) instead of 3.01(s)

Profiling a Python code

Visualization

- Profiling results can be visualized with SnakeViz
- We must be in console mode

```
1 python3 -m cProfile -o mandel.prof mandel.py
2 snakeviz --port 6542 --hostname localhost --server mandel.prof
```



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1000000	3.415	3.415e-06	4.807	4.807e-06	mandel.py:9(mandel)
24214688	1.393	5.752e-08	1.393	5.752e-08	~:0(<built-in method builtins.abs>)
1	0.2763	0.2763	5.087	5.087	mandel.py:17(mandel_set)
316	0.03232	0.0001023	0.03232	0.0001023	~:0(<built-in method builtins.compile>)

Profiling a Python code

Details at the line level

- We know that most of the time is spent in the `mandel` function
- We shall use the `line_profiler` package on this function to get details at the line level

```
1   [In] %load_ext line_profiler
2   [In] %lprun -f mandel mandel_set()
```

Which gives the result

```
Timer unit: 1e-06 s

Total time: 22.8401 s
File: <ipython-input-2-9421bc2016cb>
Function: mandel at line 5

#Line       Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           def mandel(c, maxiter):
     6    1000000     250304.0      0.3      1.1      z = c
     7   24463110    6337732.0      0.3     27.7      for n in range(maxiter):
     8   24214592    8327289.0      0.3     36.5          if abs(z) > 2:
     9     751482     201108.0      0.3      0.9              return n
    10   23463110    7658255.0      0.3     33.5          z = z*z + c
    11     248518      65444.0      0.3      0.3      return n
```

Profiling a Python code

Details at the line level

This can be done in console mode as well

```
1  import line_profiler
2
3  @profile
4  def mandel(c, maxiter):
5    z = c
6    for n in range(maxiter):
7      if abs(z) ¿ 2:
8        return n
9      z = z*z + c
10   return n
```

Then on the command line

```
1 kernprof -l -v mandel.py
```