Targeting Performance with Python

David A. Clarke

Z02 – Software Development Center

28 Jan 2022

UNIVERSITÄT BIELEFELD
Fakultät für Physik

CRC–TR 211
Strong-interaction matter under extreme conditions

Foreword

The purpose of this talk is to

- start thinking about computational speed in Python[1]
- and learn a little about parallelization.

I assume you can already do some scientific programming and that you attended this year's introductory Python talk. It will also help to have attended the OOP talk.

This talk will be uploaded to Redmine, where you can find other excellent talks by the Software Development Center.

If you have questions or comments, please **interrupt me!**

[1] As usual, we use Python3.

Outline

1. How fast is Python?
2. Parallelization strategies
3. Wrap-up

1. How fast is Python?
2. Parallelization strategies
3. Wrap-up

Thinking about time complexity

One way to judge speed of algorithm is time complexity. Express run time as function of input size. Since run times are

- negligible for small input size and
- generally difficult to compute exactly as function of input size

we focus on asymptotic behavior. Usually expressed as

$$\text{operation} \sim \mathcal{O}(f(n))$$

for some function f of size n.

Some example time complexities

Commonly used complexities[2] for lists and dictionaries:

Operation	Complexity
append	$\mathcal{O}(1)$
length	$\mathcal{O}(1)$
store	$\mathcal{O}(1)$
construct	$\mathcal{O}(\text{len})$
iterate	$\mathcal{O}(n)$
`in`	$\mathcal{O}(n)$

You can find more complexities here.

[2] It should be noted these are worst case estimates. For example when using `in` to search these objects, it turns out that dictionaries can be much faster than lists, and have an average complexity $\mathcal{O}(1)$. This is because dictionaries are implemented using hash tables.

Timing your code

This can be done straightforwardly with the `time` class:

```
import time

t0 = time.time()

# some stuff...

t1 = time.time()

print("Took", t1-t0, "[s]")
```

```
Took 7.588989019393921 [s]
```

For measuring small bits of code, `timeit` may be better alternative:

```
python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 5: 30.2 usec per loop
```

Using `numpy`

As mentioned in the introductory talk , using built-in objects and methods from `numpy` can give you a significant performance boost. Make sure you are using

- `numpy` **arrays**, which are localized in memory
- **built-in functions**, which are parallel & implemented in C
- same with **element-wise operations** for arrays

Some detailed analysis here gives a better idea of what one has to gain from using `numpy`.

1. How fast is Python?
2. Parallelization strategies
3. Wrap-up

One strategy: `numba`

In some situations, one can get a performance boost with `numba`[3]. The idea is to use a just-in-time (JIT) compiler, i.e. the code is compiled automatically right before you run it.

Presumably you don't have this. (At least I didn't.) Install it with

```
pip install numba
```

[3] Some simple examples using `numba` are here.

One strategy: numba

Using numba is accomplished through a decorator, which is basically a function that takes a function as argument and returns a function.

```
from numba import jit

# nopython = True : Do not use Python interpreter
# nopython = False: More flexible but slower

@jit(nopython=True) # jit decorator
def decorated_fuction():
  # Do some stuff...
```

So I just decorate and I'm done? Well, no. Best for functions that

1. mostly consist of math operations
2. with numpy[4]
3. and lots of loops

[4] Note that not all numpy features are supported, however. A summary of what's supported can be found here. In this case, functions may need to be implemented in raw Python.

A basic example

Let's try one of the examples from here :

```
from numba import jit
import random
import numpy as np


@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Another strategy: `concurrent.futures`

`concurrent.futures`[5] gives one a lot of control over parallelization. We use a pool of threads/processes, i.e. a set of threads/processes always ready for use[6]. One creates an executor object whose type depends on one whether would like a pool of threads or processes.

```
import concurrent.futures

executor=ThreadPoolExecutor(max_workers=8)
executor=ProcessPoolExecutor(max_workers=8)
```

After picking an executor, one uses `map` to delegate tasks:

```
def functionWithOneArgument ( argument ):
...

thingsFunctionRunsOver = [ ... ]

executor.map( functionWithOneArgument , thingsFunctionRunsOver )
```

[5] Documentation can be found here .

[6] Maintaining a pool saves time creating/destroying threads/processes for short tasks.

Threads or processes?

Each process gets its own memory space, while threads of a process generally share memory. Which to use?

- Multiple threads are shared by a single CPU.
- For lightweight tasks, one CPU can easily execute two threads simulataneously.
- For heavy tasks, one thread might spend CPU's entire resources.

I always parallelize using processes[7], since my slowest stuff is mathy and I have easy access to machines with many processors.

[7] You may want to play around with it, it's not necessarily the best strategy. Mileage may vary.

A basic example

```
import concurrent.futures

# Let's add these arrays
a = [1,0,1,0,1,0,1,0,1]
b = [0,1,0,1,0,1,0,1,0]

indices = range(len(a))

# Accomplished with loops.
c = [0,0,0,0,0,0,0,0,0]
for i in indices:
  c[i] = a[i] + b[i]


def AplusB(i):
  return a[i] + b[i]

# Accomplished with concurrent.futures
with concurrent.futures.ProcessPoolExecutor(max_workers=8) as executor:
  c_parallel = executor.map( AplusB, indices )
c_parallel = list(c_parallel)
```

```
         c = [1, 1, 1, 1, 1, 1, 1, 1, 1]
c_parallel = [1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Difficulty: Function has multiple arguments

Okay, but what if my function needs another argument?

```
import concurrent.futures

def raiseToPower(x, n):
  return x**n

baseNumbers=[1,2,3,4,5,6,7,8]

with concurrent.futures.ProcessPoolExecutor(max_workers=8) as executor:
  raisedNumbers = executor.map( ... ? )
```

Possible solutions: Straightforward

Pass `n` in secret:

```
n=3
def raiseToPowerSecret(x):
  return x**n
```

Be content with `n`'s default value:

```
def raiseToPowerDefault(x, n=2):
  return x**n
```

Create a wrapper:

```
def raiseToPowerWrapped(x):
  return raiseToPower(x,4)
```

Pass `map` an argument array of the same size[8]:

```
with concurrent.futures.ProcessPoolExecutor(max_workers=8) as executor:
  raisedNumbers = executor.map( raiseToPower, baseNumbers, [2,2,2,2,2,2,2,2] )
raisedNumbers = list(raisedNumbers)
```

[8] Thanks to Volodymyr for pointing out this possibility.

Possible solutions: Advanced

Use a class!

```
class powerRaiser:

  def __init__(self, x, n):
    self._x = x
    self._n = n
    with concurrent.futures.ProcessPoolExecutor(max_workers=8) as executor:
      result = executor.map( self.raiseToPowerClass, self._x )
    self._result = list(result)

  def raiseToPowerClass(self, x):
    return x**self._n

  def getResult(self):
    return self._result


pr = powerRaiser(baseNumbers, 2)
raisedNumbers = pr.getResult()
print(raisedNumbers)
```

Real-life jackknife example

```
class nimbleJack:

    """ Class allowing for parallelization of the jackknife function. """

    def __init__(self, func, data, nblocks, confAxis, return_sample, args, cov,
        parallelize, nproc):

        self._func=func
        self._data=np.array(data)
        ...
        if parallelize:
            with concurrent.futures.ProcessPoolExecutor(max_workers=nproc) as
                    executor:
                blockval=executor.map(self.getJackknifeEstimator, blockList)
        ...

    def getJackknifeEstimator(self,i):
        """ Gets ith estimator from throwing away jackknife block i. """
        ...


    def getResults(self):
        return self._mean, self._error


def jackknife(func, data, numb_blocks, conf_axis=1, return_sample, args=(),
              cov=False, parallelize=True, nproc=8):
    jk = nimbleJack(func, data, numb_blocks, conf_axis, return_sample, args,
                    cov, parallelize, nproc)
    return jk.getResults()
```

1. How fast is Python?
2. Parallelization strategies
3. Wrap-up

Summary

When thinking about Python code:

- Be strategic about loop placement
- Use `numpy`'s built-ins
- Try parallelizing with `numba`
- Otherwise try with `concurrent.futures`
- Use `time` to see how well you did!

Thanks for listening!