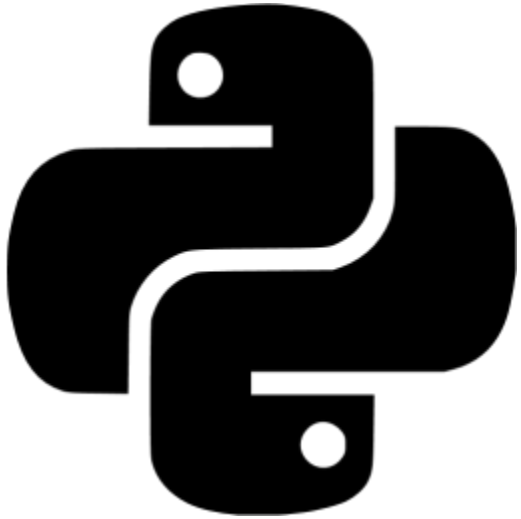
Multiprocessing in Python



Amir (@_ambodi)

Stockholm Python Meetup (04/03/2015)

amir@gapminder.org

 : ambodi

 : @_ambodi

- **Sentinel** (Real-time Social Media Analytics): www.github.com/ambodi/sentinel
- **Tokyoholm**: tokyoholm.tumblr.com

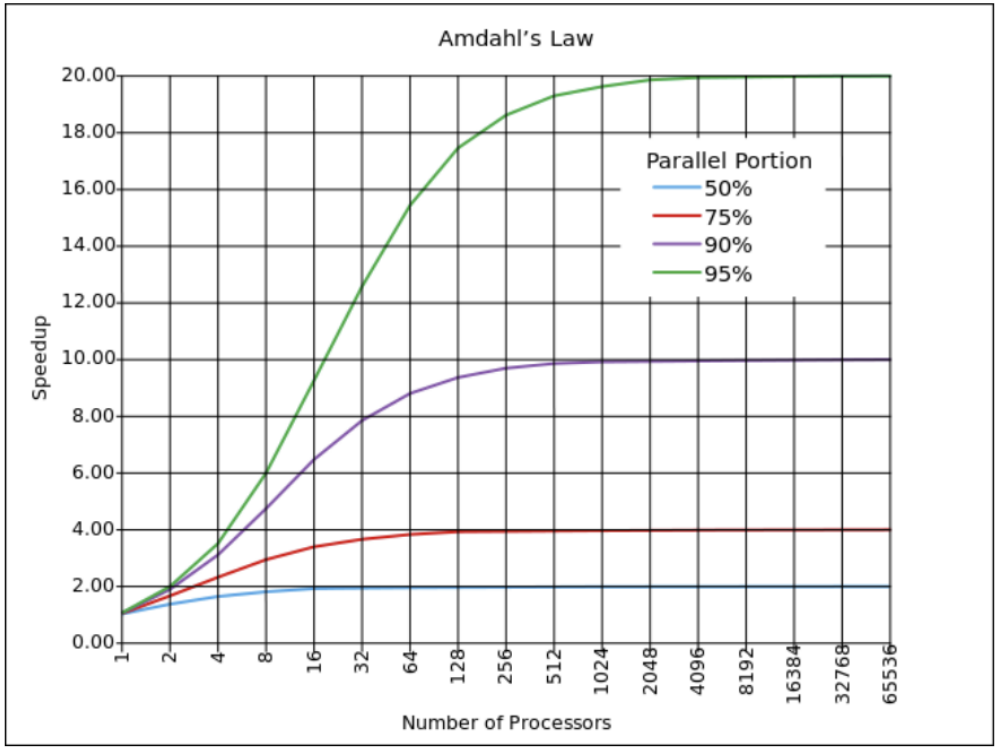
Q & A

Q: So now data centers are being filled with shiny new computers and the top-end machines have as many as 24 cores. But what about performance? Are these shiny new machines going 24 times faster?

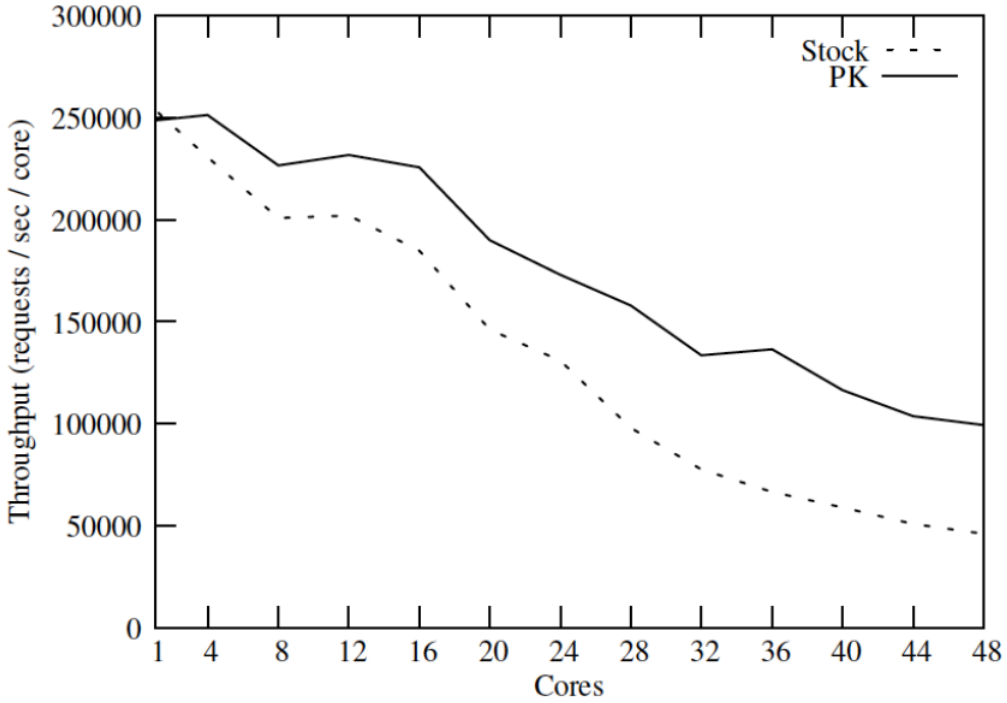
A: For some problems yes - but for many problems no.

Conclusion: The under-utilization of the CPU is a serious problem.

http://joearms.github.io/2013/03/28/solving-the-wrong-problem.html


Amdahl's Law
Speedup
Number of Processors
Parallel Portion
50%
75%
90%
95%
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
16.00
18.00
20.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

Amdahl's Law



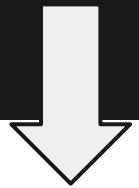
Source: http://pdos.csail.mit.edu/papers/linux:osdi10.pdf

Memcache Throughput While Scaling

Part I

Multi-threading

Example



```
def rand_string(length, pos, queue):
    rand_str = ''.join(random.choice(
        string.ascii_lowercase
        + string.ascii_uppercase
        + string.digits) for i in range
(length))
    queue.put((pos, rand_str))
```

Multi-threading

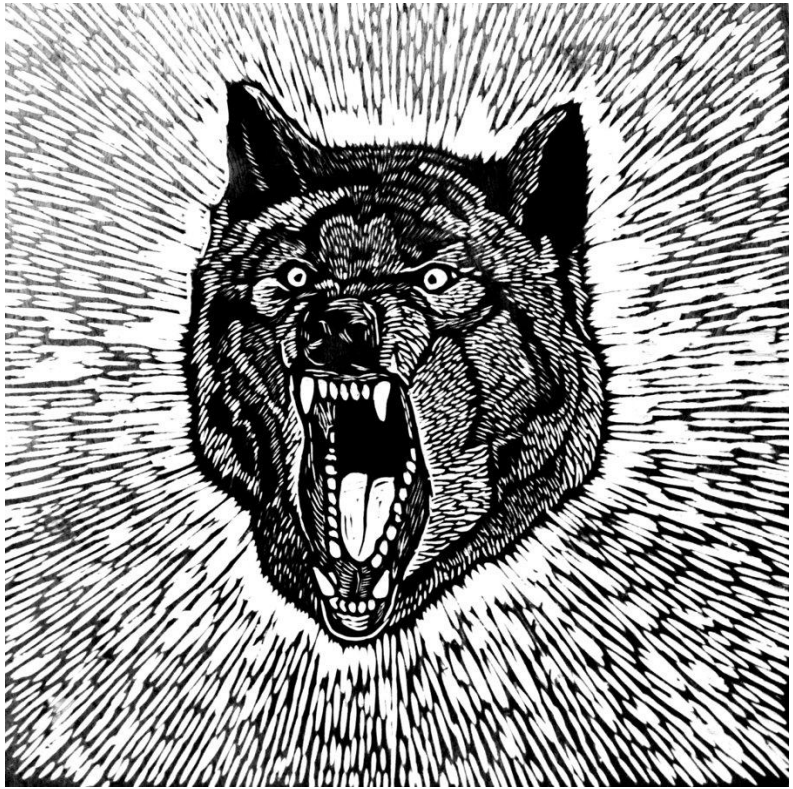
- **`_module()`** provides low-level primitives for working with multiple threads
- For synchronization, simple locks (also called mutexes or binary semaphores) are provided
- For systems lacking the **`_thread`** module, the **`_dummy_thread`** module is available

/thread/example_threading.py

```
threads = [Thread(target=rand_string, args=(5, x, output))
for x in range(total_num_threads)]

# Run processes
for p in threads:
    p.start()

# Exit the completed processes
for p in threads:
    p.join()
```

/thread/serial_multi_thread.py

```
Amirs-MacBook:thread amir$  python -mtimeit -s'import
serial_multi_thread' 'serial_multi_thread.
rand_string_serial()'
10000 loops, best of 3: 34.2 usec per loop

Amirs-MacBook:thread amir$  python -mtimeit -s'import
serial_multi_thread' 'serial_multi_thread.
rand_string_multi_thread()'
1000 loops, best of 3: 442 usec per loop
```



Here comes… Insanity Wolf!

Global Interpreter Lock

- Simple & Fast
- The GIL is controversial
- Only one thread executes bytecode at a time
- Implicitly safe against concurrent access.
- Easier for the interpreter to be multi-threaded
- Getting rid of the GIL is an occasional topic on the python-dev mailing list ☺ ☺ ☺

Global Interpreter Lock (II)

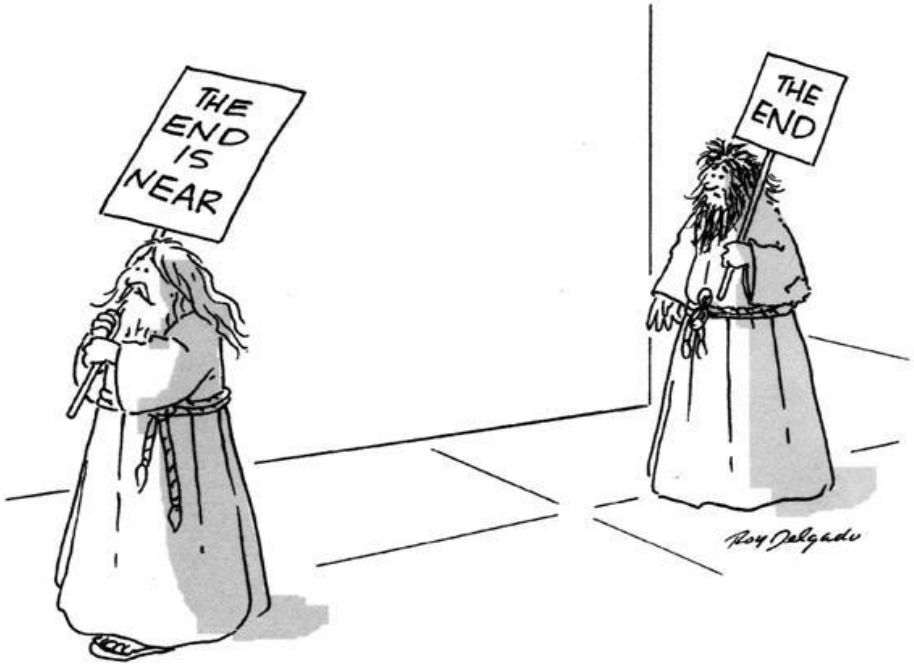
- long-running operations, such as I/O, image processing, and NumPy number crunching, happen outside the GIL.
- **only** in multithreaded programs that spend a lot of time inside the GIL, interpreting CPython bytecode, that the GIL becomes a **bottleneck**.

Python’s Hardest Problem

“For more than a decade, no single issue has caused more frustration or curiosity for Python novices and experts alike than the Global Interpreter Lock.

...Every field has one. A problem that has been written off as too difficult, too time consuming. Merely mentioning an attempt to solve it raises eyebrows. Long after the community at large has moved on, it is taken up by those on the fringe. Attempts by novices are undertaken for no other reason than the difficulty of the problem and the imagined accolades that would accompany a solution”

Jeff Knapp: http://www.jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/

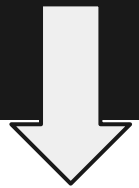


No Comment!

Part II:

`Multiprocessing`

Example



```
def rand_string(length, pos, queue):
    rand_str = ''.join(random.choice(
        string.ascii_lowercase
        + string.ascii_uppercase
        + string.digits) for i in range
(length))
    queue.put((pos, rand_str))
```

`Process` Module

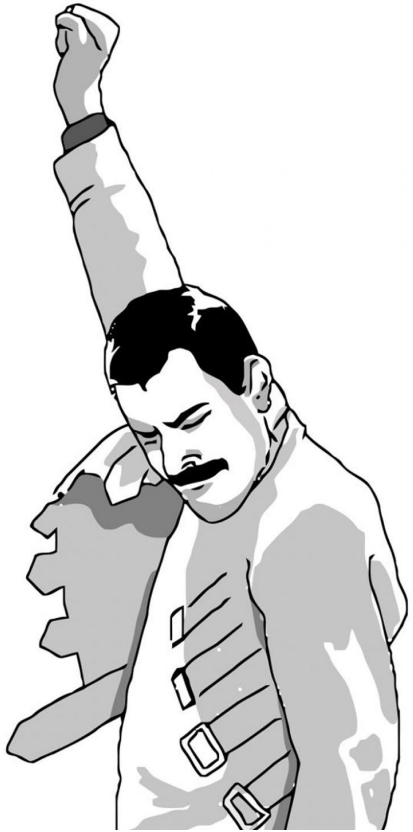
- Creating a `Process` object
- Calling its start() method.
- Follows the API of threading.Thread.

/multi_process/process/example_process.py

```
output = mp.Queue()

# Setup a list of processes that we want to run
processes = [mp.Process(target=rand_string, args=(5, x, output)) for x in range(num_of_fork_process)]

# Run processes
for p in processes:
   p.start()

# Exit the completed processes
for p in processes:
   p.join()

# Get process results from the output queue
results = [output.get() for p in processes]
results.sort()
```

/multi_process/process/serial_multi_process.py

```
Amirs-MacBook:stockholm_python_meetup_04_march amir$
python -mtimeit -s'import serial_multi_process
serial_multi_process.rand_string_serial()'
1000 loops, best of 3: 247 usec per loop

Amirs-MacBook:stockholm_python_meetup_04_march amir$
python -mtimeit -s'import serial_multi_process'
'serial_multi_process.rand_string_multiprocess()'
10000 loops, best of 3: 79.2 usec per loop
```



YES!

Buzzinga!

```
Traceback (most recent call last):
  File "/Users/amir/dev/personal/stockholm_python_meetup_04_march/process/example_process.py", line 28, in <module>
    p.start()
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/multiprocessing/process.py", line 130, in start
    self._popen = Popen(self)
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/multiprocessing/forking.py", line 121,
in __init__
    self.pid = os.fork()
OSError: [Errno 35] Resource temporarily unavailable
```

How many processes can be created with Python `os.fork`?

http://stackoverflow.com/questions/20452150/how-many-processes-can-be-created-with-python-os-fork

Why do you want to do this? – msw Dec 8 '13 at 10:32

No comment!

```
Amirs-MacBook:~ amir$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 4864
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 709
virtual memory          (kbytes, -v) unlimited
```

$ ulimit -a

http://bugs.python.org/issue19675

```
try:
    for i in range(self._processes - len(self._pool)):
        w = self.Process(target=worker,
                         args=(self._inqueue, self._outqueue,
                               self._initializer,
                               self._initargs, self._maxtasksperchild)
                        )
        self._pool.append(w)
        w.name = w.name.replace('Process', 'PoolWorker')
        w.daemon = True
        w.start()
        debug('added worker')
except:
    debug("Process creation error. Cleaning-up (%d) workers." % (len(self._pool)))

    for process in self._pool:
        if process.is_alive() is False:
            continue

        process.terminate()
        process.join()

    debug("Processing cleaning-up. Bubbling error.")
    raise
```

`Pool` Module

- Parallelizing the execution of a function across multiple input values

- Distributing the input data across processes (**data parallelism**)

/multi_process/pool/serial_multi_process.py

```
# Setup a list of processes that we want to run
p = Pool(total_num_pool)
p.map(rand_string, [(5, x, output) for x in range
(total_num_pool)])

# Get process results from the output queue
results = output.get().sort()

print results
```

/multi_process/pool/serial_multi_process.py

```
Amirs-MacBook:stockholm_python_meetup_04_march amir$
python -mtimeit -s'import serial_multi_process
serial_multi_process.rand_string_serial()'
1000 loops, best of 3: 247 usec per loop

Amirs-MacBook:stockholm_python_meetup_04_march amir$
python -mtimeit -s'import serial_multi_process'
'serial_multi_process.rand_string_pool()'
10000 loops, best of 3: 102.2 usec per loop
```

Process > Pool

- `Process` module performed better
- `Pool` **usually** performs better with more chunks of data

MADlib (http://madlib.net/)

- Operate on the data In database, directly.
- Do not move it between multiple runtime environments unnecessarily

That’s all folks! ☺