

# Concurrent Computations on Multicore Processors

## Threads & Subprocesses

Christian Zielinski

18<sup>th</sup> June 2015 – PyCon SG 2015

# Who am I?

- Currently a Ph.D. student at Nanyang Technological University
  - Working in computational physics, i.e. simulation of elementary particles with Monte Carlo methods (“lattice quantum chromodynamics”)
- First experiments with Python around ten years ago
- Heavy user of Python, **numpy** and **scipy** for data analysis and evaluation of numerical algorithms
- Slides available on:

[www.github.com/czielinski/pyconsg2015](http://www.github.com/czielinski/pyconsg2015)

# Why concurrency?

- Python is *awesome*, but slow
  - C/C++ is typically 10–100 times faster
- Single CPU core performance only improves slowly over time
  - But modern processors have increasing number of cores
- Blocking operations like I/O cause idle time
  - Rather do computations in the meantime
- Concurrency helps to mitigate all problems
  - Potentially significant speed ups

# Concurrency in Python

- How to make use of concurrency in Python?
- For stock Python typically use either
  - Threads via the `threading` module
  - Subprocesses via the `multiprocessing` module
- They provide convenient high-level interfaces
  - `threading` and `multiprocessing` have similar API
- There are great third-party modules to go beyond a single CPU:
  - `mpi4py`, `pycuda`, `pyopencl` (but not today's topic)

# Global Interpreter Lock (GIL)

- CPython's threads are real POSIX / Windows threads
- The Global Interpreter Lock (GIL) prevents that more than one thread is executing Python bytecode at the same time
- The Global Interpreter Lock:
  - makes the CPython interpreter explicitly thread-safe
  - simplifies CPython's code base
  - prevents true parallel threading
  - is released during I/O
- The GIL is implementation dependent
  - CPython and PyPy have a GIL
  - Jython and IronPython have no GIL

# Threads

- Threads are accessible via `threading` module
  - Can be spawned quickly
  - Share memory
- Threads can be useful for I/O bound problems
  - Do useful operations during blocking operations
- Usually not suited for CPU bound problems
  - GIL prevents true parallel computations (with a few exceptions, e.g. some `numpy` functions)
  - Introduce additional overhead
- Use `Lock` to define critical sections, i.e. code that requires mutual exclusion of access

## Simple threading example

```
1 import threading as th
2
3 def worker(lock, num):
4     cubed = num**3
5     # A worker should actually avoid I/O ...
6     lock.acquire()
7     print("{} cubed is {}".format(num, cubed))
8     lock.release()
9
10 my_threads = []
11 for i in range(4):
12     lock = th.Lock()
13     t = th.Thread(target=worker, args=(lock, i))
14     my_threads.append(t)
15     t.start()
```

# Subprocesses (I)

- Subprocesses accessible via `multiprocessing` module
  - Self-sufficient interpreter instance
  - No memory sharing
- Subprocesses have their own interpreter instance
  - Spawning takes long time, significant overhead
  - Every subprocess has own GIL, allow for true parallelism



## Subprocesses (II)

- Subprocesses suited for CPU bound problems
  - Can do computations truly in parallel to utilize several cores
  - Usually not useful to spawn more than one subprocess per *physical* core
- Not suited for I/O bound problems
  - Cannot e.g. read files faster from HDD
  - To keep cores busy need to feed data quickly enough
  - Potential memory bandwidth bottleneck for large number of cores

## Simple multiprocessing example

```
1 import multiprocessing as mp
2
3 def worker(lock, num):
4     cubed = num**3
5     # A worker should actually avoid I/O ...
6     lock.acquire()
7     print("{} cubed is {}".format(num, cubed))
8     lock.release()
9
10 my_procs = []
11 for i in range(4):
12     lock = mp.Lock()
13     p = mp.Process(target=worker, args=(lock, i))
14     my_procs.append(p)
15     p.start()
```

# Scaling and Amdahl's law

- Using  $N$  cores does not automatically reduce runtime by a factor of  $N$ 
  - Some code section inherently serial, like I/O
  - Some algorithms are difficult to parallelize, e.g. recursive functions
  - One usually parallelizes only the performance-critical parts of the code
- Amdahl's law [Amdahl '67]
  - Parallelizing a program with serial runtime  $T(1)$  with  $N$  processes and serial code fraction  $f \in [0, 1]$  results in runtime:

$$T(N) = T(1) \cdot \left( f + \frac{1}{N}(1 - f) \right)$$

- So for sufficiently large  $N$  runtime is dominated by serial code sections!

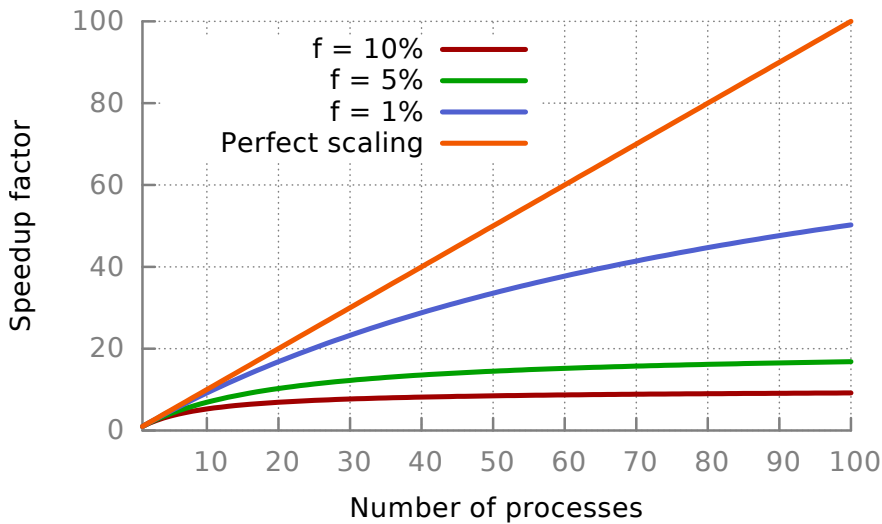


Figure: Amdahl's law

## Some general remarks

- Split up larger problems into smaller subproblems with appropriate size
  - If too small, performance degraded due to overhead
  - If too large, load balancing becomes problematic
- Many methods are not thread-safe
  - Use `Lock.acquire()` and `Lock.release()` in critical code sections
  - Note: even `print` is not thread-safe
- Arguing about the correctness of a parallel program is hard
  - Order of computations are not fixed

# API of the multiprocessing module

## Worker pools

- Simplest parallelization strategy is to use a worker pool
  - Accessible via `multiprocessing.Pool`
- Important methods:
  - `map` (`map_async`) for (asynchronous) parallel map
  - `apply_async` for asynchronous function evaluation
- Limitations:
  - Can only deal with pickable objects
  - Cannot use `lambda` expressions, nested functions, class methods
  - Can use global functions, global classes and partial functions via `functools.partial`
  - `Pool` will not work in interactive sessions  
(`__main__` module has to be importable by the children)

## Parallel map

```
1 from multiprocessing import Pool
2
3 def cube(num):
4     return num**3
5
6 if __name__ == '__main__':
7     # Can also specify 'processes' parameter
8     p = Pool()
9     # Equivalent of serial map(cube, range(10))
10    res = p.map(cube, range(10))
11    print(res)
12    # Output: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Note: This overly simplified example runs slower than the serial version



## Asynchronous function evaluation

```
1 from multiprocessing import Pool
2
3 def double(arg):
4     return 2*arg
5
6 def cube(num):
7     return num**3
8
9 if __name__ == '__main__':
10     p = Pool()
11     f1_get = p.apply_async(cube, args=(6,))
12     f2_get = p.apply_async(double, args=('haha',))
13     # Do other work here
14     res = [f1_get.get(), f2_get.get()]
15     print(res)
16     # Output: [216, 'hahahaha']
```

# Sharing data

## Shared memory maps

To share state with a `Process` use shared memory maps:

```
1 from multiprocessing import Process, Array
2
3 def cube_arr(arr):
4     for i in range(len(arr)):
5         arr[i] = arr[i]**3
6
7 if __name__ == '__main__':
8     shared_arr = Array('i', range(10)) # 'd' for double
9     p = Process(target=cube_arr, args=(shared_arr,))
10    p.start()
11    # Do other work here
12    p.join()
13    print(shared_arr[:])
14    # Output: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

## Server processes and managers

Alternatively use a **Manager** for more complex objects:

```
1 from multiprocessing import Process, Manager
2
3 def modify_names(names):
4     names['Alice'] = 42
5     if 'Bob' in names:
6         shared_dict['Bob'] += 1
7
8 if __name__ == '__main__':
9     with Manager() as m:
10         shared_dict = m.dict()
11         shared_dict['Bob'] = 7
12         p = Process(target=modify_names, args=(shared_dict,))
13         p.start()
14         # Do other work here
15         p.join()
16         print(shared_dict) # Output: {'Bob': 8, 'Alice': 42}
```

# Interprocess communication

# Pipes

To communicate between two processes use a Pipe:

```
1 from multiprocessing import Process, Pipe
2
3 def cube(my_pipe):
4     data = my_pipe.recv()
5     my_pipe.send([n**3 for n in data])
6
7 if __name__ == '__main__':
8     parent_pipe, child_pipe = Pipe()
9     p = Process(target=cube, args=(child_pipe,))
10    p.start()
11    parent_pipe.send(range(10))    # Send work
12    # Do other work here
13    print(parent_pipe.recv())
14    p.join()
15    # Output: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

## Queues

For several processes can use a Queue:

```
1 from multiprocessing import Process, Queue
2
3 def place_work(q):
4     q.put(range(10))
5
6 def process_work(q):
7     data = q.get()
8     q.put([n**3 for n in data])
9
10 if __name__ == '__main__':
11     queue = Queue()
12     p1 = Process(target=place_work, args=(queue,))
13     p2 = Process(target=process_work, args=(queue,))
14     p1.start(); p2.start()    # Now do other work
15     p1.join(); p2.join()
16     print(queue.get())    # Output: [1, 64, 343]
```

## Guidelines (I)

- Most important: Only parallelize where necessary!
- Try to avoid sharing state if possible
- If need to share:
  - Use **Value** and **Array** for simple data
  - Use server process **Manager** for more complex objects
- For interprocess communication use **Pipe** and **Queue**
  - Pipes for fast point-to-point communication
  - Queues are multi-producer, multi-consumer FIFO queues



## Guidelines (II)

- Explicitly pass resources to subprocesses and avoid global variables as they can lead to inconsistencies (Windows)
- Ensure that the main module can be safely imported by a new Python instance (Windows)
  - Use `if __name__ == '__main__':` guard to avoid side effects
- Avoid `Process.terminate` as it might break access to shared resources
- Read more on:  
<https://docs.python.org/2/library/multiprocessing.html#programming-guidelines>

# Time for a short demo!

- Midpoint rule for numerical integration

$$\int_a^b f(x) dx \approx h \sum_{k=1}^N f\left(a + \left(k - \frac{1}{2}\right)h\right), \quad h = \frac{b-a}{N}$$

- Allows for straightforward parallel evaluation

$$\int_a^b f(x) dx = \int_a^c f(x) dx + \int_c^b f(x) dx, \quad \forall c \in \mathbb{R}$$

- Can split up large integration range into several small ones, which can be evaluated in parallel

# Questions & Answers

(if time permits)

Slides on:

`www.github.com/czielinski/pyconsg2015`