Concurrent Computations on Multicore Processors

Threads & Subprocesses

Christian Zielinski

18th 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”)
- Heavy user of Python, **`numpy`** and **`scipy`** for data analysis and evaluation of numerical algorithms
- First experiments with Python around ten years ago
- Slides available on:

`www.github.com/czielinski/pyconsg2015`

Why concurrency?

- Python is *awesome*, but slow (if you don't use C/C++ extensions)
- 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 us 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)

- 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
- Read more on `https://wiki.python.org/moin/GlobalInterpreterLock`

Threads

- Threads are accessible via **`threading`** module
  - Can be spawned quickly
  - Share memory
- CPython's threads are real POSIX/Windows threads
- Threads can be useful for I/O heavy 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 a **`Lock`** to define critical sections,
  i.e. code that requires mutual exclusion of access

Simple threading example

```
import threading as th

def worker(lock, num):
  cubed = num**3
  # A worker should actually avoid I/O ...
  with lock:
    print("{} cubed is {}".format(num, cubed))

lock = th.Lock()
my_threads = []
for i in range(4):
  t = th.Thread(target=worker, args=(lock, i))
  my_threads.append(t)
  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

```python
import multiprocessing as mp

def worker(lock, num):
  cubed = num**3
  # A worker should actually avoid I/O ...
  with lock:
    print("{} cubed is {}".format(num, cubed))

lock = mp.Lock()
my_procs = []
for i in range(4):
  p = mp.Process(target=worker, args=(lock, i))
  my_procs.append(p)
  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 using N processes and a serial code fraction $f \in [0, 1]$ (by runtime) results in a runtime of:

$$T_N = T_1 \cdot \left(f + \frac{1}{N} (1 - f) \right)$$

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

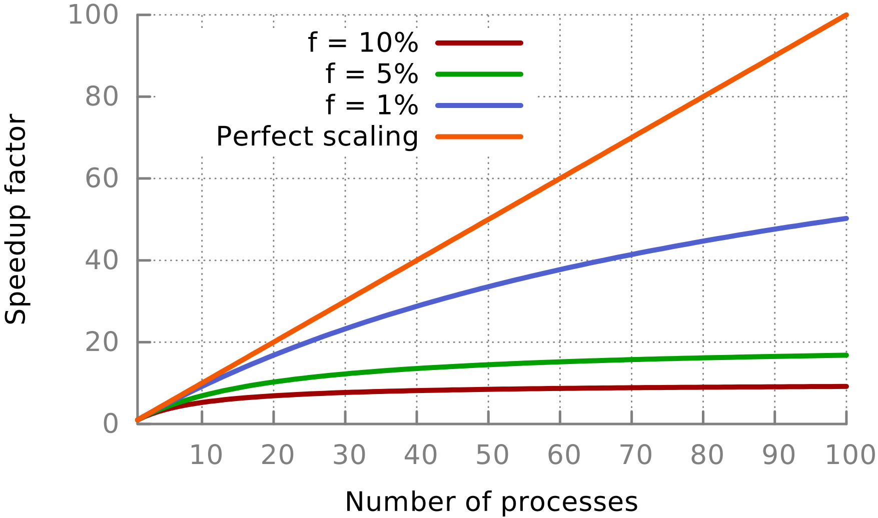

Figure: Amdahl's law

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
- 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

```python
from multiprocessing import Pool

def cube(num):
  return num**3

if __name__ == '__main__':
  # Can also specify 'processes' parameter
  p = Pool()
  # Equivalent of serial map(cube, range(10))
  res = p.map(cube, range(10))
  print(res)
  # 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

```
from multiprocessing import Pool

def double(arg):
  return 2*arg

def cube(num):
  return num**3

if __name__ == '__main__':
  p = Pool()
  f1_get = p.apply_async(cube, args=(6,))
  f2_get = p.apply_async(double, args=('haha',))
  # Do other work here
  res = [f1_get.get(), f2_get.get()]
  print(res)
  # Output: [216, 'hahahaha']
```

Sharing data

Shared memory maps

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

```
from multiprocessing import Process, Array

def cube_arr(arr):
  for i in range(len(arr)):
    arr[i] = arr[i]**3

if __name__ == '__main__':
  shared_arr = Array('i', range(10))  # 'd' for double
  p = Process(target=cube_arr, args=(shared_arr,))
  p.start()
  # Do other work here
  p.join()
  print(shared_arr[:])
  # Output: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Server processes and managers

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

```
from multiprocessing import Process, Manager

def modify_names(names):
  names['Alice'] = 42
  if 'Bob' in names:
    shared_dict['Bob'] += 1

if __name__ == '__main__':
  with Manager() as m:
    shared_dict = m.dict()
    shared_dict['Bob'] = 7
    p = Process(target=modify_names, args=(shared_dict,))
    p.start()
    # Do other work here
    p.join()
    print(shared_dict)  # Output: {'Bob': 8, 'Alice': 42}
```

Interprocess communication

Pipes

To communicate between two processes we use a **Pipe**:

```
from multiprocessing import Process, Pipe

def cube(my_pipe):
  data = my_pipe.recv()
  my_pipe.send([n**3 for n in data])

if __name__ == '__main__':
  parent_pipe, child_pipe = Pipe()
  p = Process(target=cube, args=(child_pipe,))
  p.start()
  parent_pipe.send(range(10))  # Send work
  # Do other work here
  print(parent_pipe.recv())
  p.join()
  # Output: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Queues

For several processes we can use a **Queue**:

```
from multiprocessing import Process, Queue

def place_work(q):
  q.put(range(10))

def process_work(q):
  data = q.get()
  q.put([n**3 for n in data])

if __name__ == '__main__':
  queue = Queue()
  p1 = Process(target=place_work, args=(queue,))
  p2 = Process(target=process_work, args=(queue,))
  p1.start(); p2.start(); p1.join(); p2.join()
  print(queue.get())
  # Output: [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

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

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

- Allows for straightforward parallel evaluation

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

- 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`