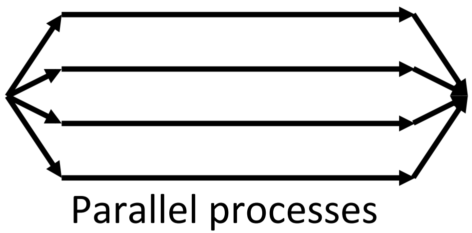
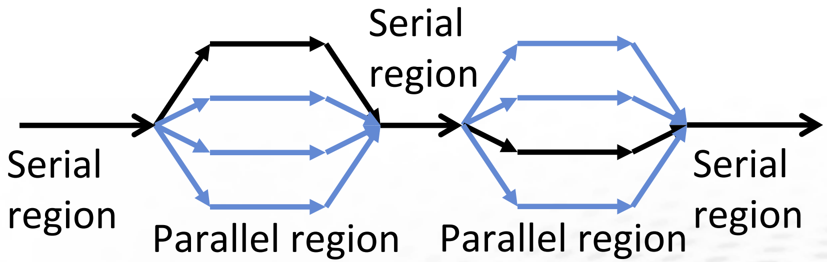
MULTIPROCESSING

– PROCESS BASED ”THREADING”

Processes and threads





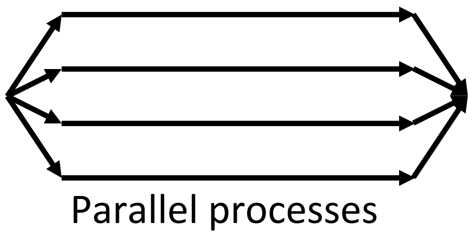
Process

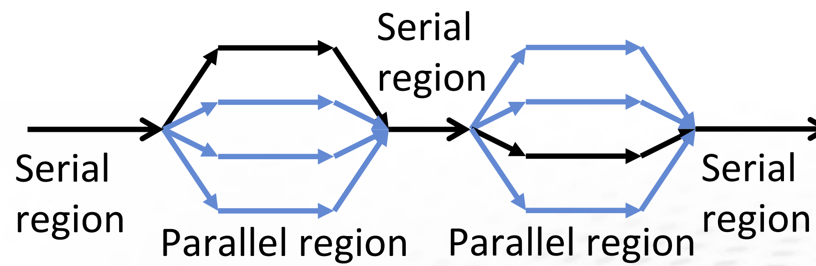
- Independent execution units
- Have their own state information and *own address spaces*

Thread

- A single process may contain multiple threads
- Have their own state information, but share the address space of the process

Processes and threads





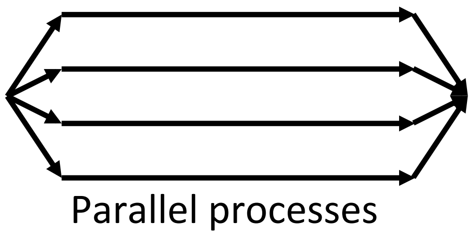
Process

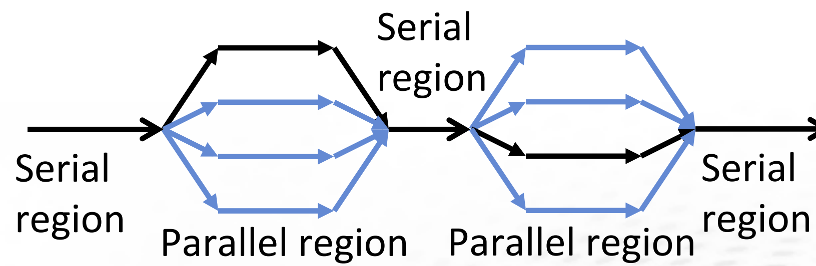
- Long-lived: spawned when parallel program started, killed when program is finished
- Explicit communication between processes

Thread

- Short-lived: created when entering a parallel region, destroyed (joined) when region ends
- Communication through shared memory

Processes and threads





Process

- MPI
  - good performance
  - scales from a laptop to a supercomputer
- multiprocessing module
  - relies on OS for forking
  - limited communication

Thread

- OpenMP
  - C / Fortran, not Python
- threading module
  - only for I/O bound tasks (maybe)

Multiprocessing

- Underlying OS used to spawn new independent subprocesses

- Communication possible only through dedicated, shared communication channels
  - Queues, Pipes
  - must be created before a new process is forked

Spawn a process

spawn.py

```
from multiprocessing import Process
import os

def hello(name):
    print 'Hello', name
    print 'My PID is', os.getpid()
    print "My parent's PID is", os.getppid()

# Create a new process
p = Process(target=hello, args=('Alice', ))
p.start() # start the process
p.join()  # end the process

print 'Spawned a new process from PID', os.getpid()
```

Synchronisation

- Processes are independent and execute code in an asynchronous manner
  - no guarantee on the order of execution
- Explicit synchronisation can be forced by the user

lock.py

```
from multiprocessing import Process, Lock

def hello(lock, id):
    lock.acquire()
    print 'Hello world! My ID is', id
    lock.release()

lock = Lock()
for i in range(10):
    Process(target=hello, args=(lock, i)).start()
```

Communication

- Sharing data
  - shared memory, data manager
- Pipes
  - direct communication between two processes
- Queues
  - work sharing among a group of processes
- Pool of workers
  - offloading tasks to a group of worker processes

Shared memory

- Shared memory similar to Direct Memory Access (DMA) possible
  - multiprocessing.Value
  - multiprocessing.Array

shared-mem.py

```
def squared(a):
    for i in range(len(a)):
        a[i] = a[i] * a[i]

numbers = Array('i', range(10))
p = Process(target=squared, args=(numbers, ))
p.start()
p.join()

print numbers[:]
```

Note:

```
def f(n):
    n.value = 3.3

n = Value('d', 0.0)
...
```

Data manager

- Data can also be shared by using a manager
  - a server process has the data and allows others to manipulate it
  - supports arbitrary Python objects
  - a single manager can be shared over the network

- Safer alternative to shared memory, but is slower due to extra overhead

Data manager

manager.py

```
from multiprocessing import Process, Manager

def f(x):
    x['Apple'] = 0.70
    x['Orange'] = 1.20

manager = Manager()
fruits = manager.dict()

p = Process(target=f, args=(fruits, ))
p.start()
p.join()

print fruits
```

Pipes

- Connection between two processes
  - data can flow in either direction
- Two connection objects that represent the two ends of the pipe
  - `send()` and `recv()` methods for sending and receiving data
  - only one process at a time can read/write safely to one end of a pipe

Pipes

pipe.py

```
from multiprocessing import Process, Pipe

def f(pipe):
    pipe.send({'Apple': 0.70, 'Orange': 1.20})
    pipe.close()

left, right = Pipe()

p = Process(target=f, args=(right, ))
p.start()
print left.recv()
p.join()
```

Queues

- FIFO (*first-in-first-out*) task queues that can be used to distribute work among processes
- Shared among all processes
  - all processes can add and retrieve data from the queue
- Automatically takes care of locking, so can be used safely with minimal hassle

Queues

queue.py

```
from multiprocessing import Process, Queue

def f(q):
    x = q.get()
    print x**2

q = Queue()
for i in range(10):
    q.put(i)

# task queue: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for i in range(10):
    p = Process(target=f, args=(q, ))
    p.start()
```

Pool of workers

- Group of processes that carry out tasks assigned to them

- Master process submits tasks to the pool
- Pool of worker processes perform the tasks (asynchronously)
- Master process retrieves the results from the pool

- Blocking and non-blocking calls available

Pool of workers

pool.py

```
from multiprocessing import Pool

def f(x):
    return x**2

pool = Pool(8)

# Blocking execution (with a single process)
result = pool.apply(f, (4,))
print result.get()

# Non-blocking execution "in the background"
result = pool.apply_async(f, (12,))
print result.get(timeout=1)
```

Pool of workers

pool-map.py

```
from multiprocessing import Pool
import time

def f(x):
    return x**2

pool = Pool(8)

# calculate x**2 in parallel for x in 0..9
print pool.map(f, range(10))

# non-blocking alternative
result = pool.map_async(f, range(10))
while not result.ready():
    time.sleep(1)
print result.get()
```

Multiprocessing summary

- Parallelism achieved by launching new OS processes
- Limited communication possible
  – shared memory, data manager
  – queues, pool of workers
- Non-blocking execution available
  – do something else while waiting for results

- Further information:
  https://docs.python.org/2/library/multiprocessing.html

Martti Louhivuori // CSC – IT Center for Science Ltd.
Python in High-Performance Computing
April 21-22, 2016 @ University of Oslo