02616

Large-scale Modelling

a look into Python

Python multiprocessing

- Bypasses GIL problems (synchronization locks)
  - Each process has its own memory space, the GIL
    en
    ac
- All c
  spaw
- Good fo

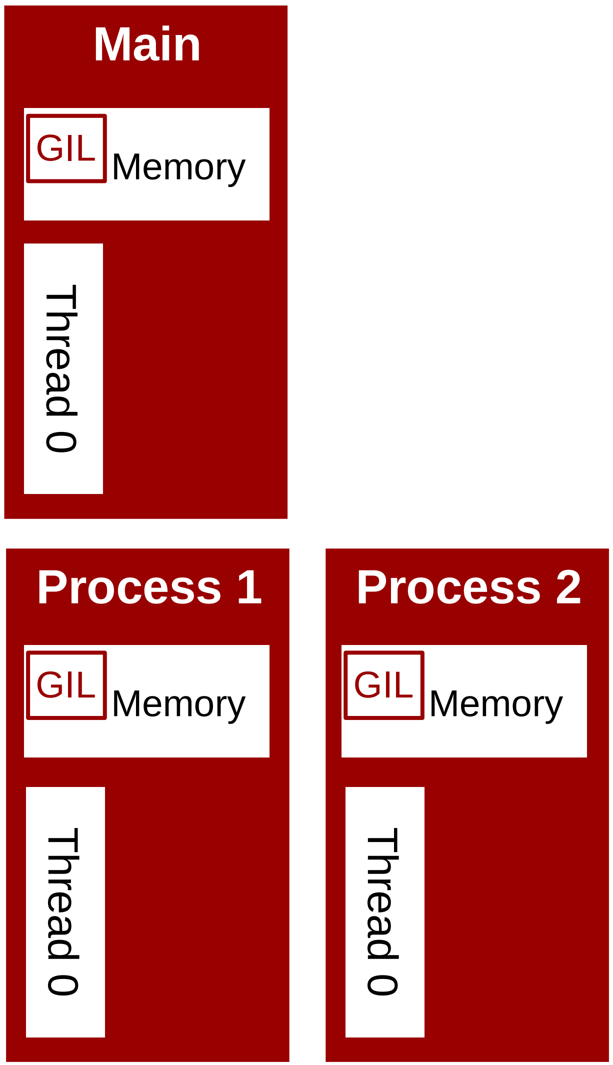
`multiprocessing` is low-level
`concurrent.futures.*` is the high-level
We'll only do low-level!

```
with multiprocessing.Pool(4) as pool:
    rets = pool.map(hard_work, range(20))
```

Calls `hard_work` 20 times, with arguments 0..19 (think `np.array_split`!)

- Can have substantial overhead!

How multiprocessing works



Pro:

Processes can always run in parallel

Cons:

Processes have more overhead

No shared memory – must explicitly **copy**

Launching processes

Pool:

```
with mp.Pool(4) as pool:
```

Pros:
- Simple to use
- Context cleans up after use
- Easily allowing return

Cons:
- Hard to customize for un-even tasks
- With tasks > NCPU’s it can be hard to use communication channels

Process:

```
p1 = mp.Process(…)
p2 = mp.Process(…)
```

Pros:
- Full control
- Simple communication through manual configuration

Cons:
- Requires manual work for return values
- Manual context clean-up

Python communication

- Communication is *hard*

  - Using `Queue`
    Simple, but non-deterministic.

```
with mp.Pool(4) as pool:
    queue = mp.Manager().Queue()
    pool.map(run, [queue for _ in range(20)])
while not queue.empty():
    print(queue.get())
```

  - Using `Pipe`
    Explicit communication between two processes
    (similar to MPI_Send/MPI_Recv)
    Always *blocking*!

We'll see examples in todays exercises!

Overhead

- What does it cost to launch additional processes?

```
def hard_work(a):
    …

with mp.Pool(1) as pool:
    pool.map(hard_work, [1])
```

```
queue = mp.Queue()
p = mp.Process(target=wrap,
               args=(queue,))
p.start()
queue.get()
```

```
~ % time python overhead.py
```

Pool

Walltime (total)

```
real 1.27
user 1.19
sys  0.14
```

CPU time in system calls

No multiprocessing

```
real 1.10
user 1.08
sys  0.01
```

CPU time in user process

Process

```
real 1.20
user 1.10
sys  0.11
```

```
real 0.63
user 3.29
sys  0.05
CPU 530%
```

Bandwidth

Transport of messages

We'll come back to why this is not always easy!

1. Start timing
2. Send a message
3. Stop timing once you know it has been received

Bandwidth = [message size] / [time spent]

Carefully think about how you time!

- Where should you start timing?
- Where should you end timing?
- Once you have full time, what does it *actually* time?
- How should you execute your program?

Everything uses resources!

Bandwidth

```
t0 = time()
<initialize program>
t1 = time()
<do heavy computations>
t2 = time()
<finalize program>
t3 = time()
```

6(7) different timings!

Bandwidth & latency!

Measuring bandwidth and latency is difficult!

<u>Bandwidth</u>

Can mean:
1) time of communication, excluding startup delay
2) time of communication, including startup delay

Be specific when using this term!
Note units! (Mb vs. MB)

<u>Latency</u>

Can mean:
1) the startup delay, excluding transfer
2) the communication delay, including transfer

Be specific when using this term!

$$T(s) = \mu_s + \frac{1}{B} s$$

Todays exercises

- Play with multiprocessing
- Gathering of results in a single process
- Reduction algorithms
- A bandwidth plot (be careful!)

The simplest is to use `Pool`, so start with that.
Then implement with `Process` (if you want!)

Volunteers for next week?