

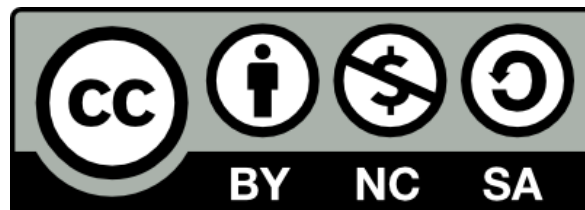
Parallel Python

Adrian Jackson

a.jackson@epcc.ed.ac.uk



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Partners



Engineering and
Physical Sciences
Research Council

Natural
Environment
Research Council



THE UNIVERSITY
of EDINBURGH



**Hewlett Packard
Enterprise**

Recap on parallelism

- Two main types of parallelism
- Shared memory parallelism
 - Single node
 - Mainly thread based
 - Single program generates set of workers (threads)
- Distributed memory parallelism
 - Can work inside a node but also enables parallelisation across nodes
 - Mainly uses message passing programming approaches (i.e. MPI)
 - Workers are individual processes (program)
 - Collaborate to reduce runtime/host large amounts of data

Parallelism in Python

- Python has a number of built in options for parallelisation

- **threading** library

- Create python threads (workers) from functions
- Can `.start()` and `.join()` them

```
import threading
```

```
def thread_function(name):
```

```
    ...
```

```
if __name__ == "__main__":
```

```
    threads = list()
```

```
    for index in range(3):
```

```
        x = threading.Thread(target=thread_function, args=(index,))
```

```
        threads.append(x)
```

```
        x.start()
```

```
    for index, thread in enumerate(threads):
```

```
        thread.join()
```

- Python global interpreter lock (GIL) restricts parallelism

- An create many threads, but only one executes at a time, regardless of available hardware

Parallelism in Python

- **multiprocessing** library
 - Enables generation of multiple process workers
 - Not restricted by the GIL

- Can create **Process**, similar to **Thread**:
`from multiprocessing import Process`

```
def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

- Share data between processes using Queues or Pipes
`from multiprocessing import Process, Queue`

```
def f(q):
    q.put([42, None, 'hello'])
if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

```
from multiprocessing import Process, Pipe
def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()
if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())    # prints "[42, None,
'hello']"
    p.join()
```

Parallelism in Python

- Multiprocessing in Python is more suited to distributed or concurrent rather than parallel computing
 - Communication mechanisms aren't as high performance
 - Don't map on to high performance interconnects
- Implicit parallelism
 - numpy provide access to parallel blas libraries
 - scipy can interface with other parallel maths libraries
- Native code interfacing
 - Call C/Fortran/C++/etc... code from Python to provide some parallel functionality
- mpi4py
 - Direct message passing interface for Python
 - <https://github.com/EPCCed/archer2-python/tree/master/lectures/dev-mpi4py>