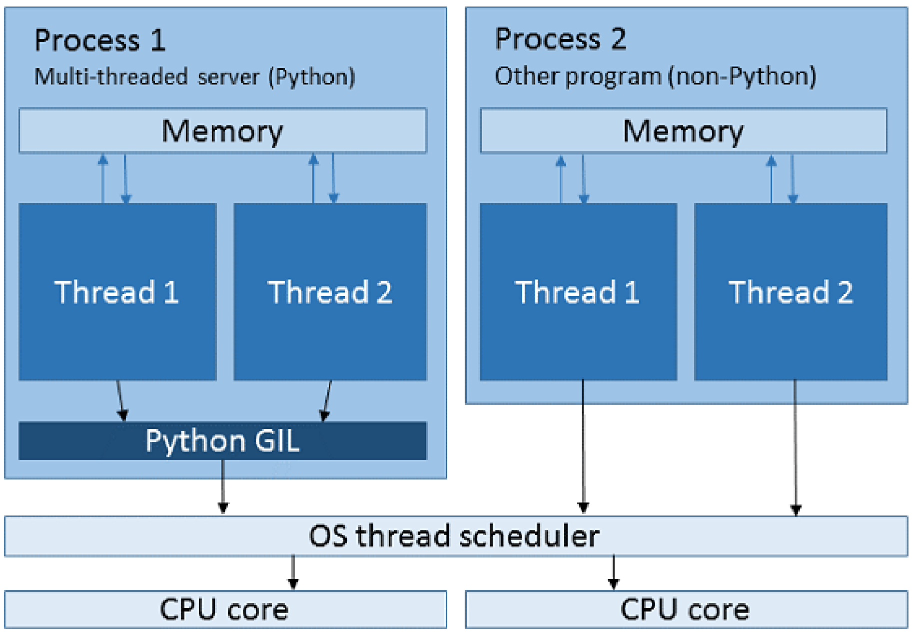
Parallel computing in **Python**

Vamshidhar Gangu

HPC specialist

Python GIL

- Global Interpreter Lock
- default Python is designed with simplicity in mind, so they made it thread-safe (GIL)
- Restrict python to run in a single thread
- **exectues only one statement at a time (serial processing or single-threading)**
- Cannot make use of data stored in shared memory



Python GIL problem

Factorial example using Threading

```
from datetime import datetime
import threading

def factorial(number):
    fact = 1
    for n in range(1, number+1):
        fact *= n
    return fact

number = 100000
thread = threading.Thread(target=factorial, args=(number,))
startTime = datetime.now()
thread.start()
thread.join()
endTime = datetime.now()
print "Time for execution: ", endTime - startTime
```

run time:

```
* 1 Thread  : 3.4 sec
* 2 Threads : 6.2 sec
```

- You don’t get the concurrency needed with Python multithreading because of the Global interpreter lock

multi-threading vs. multi-processing

multi-threading

- jobs pictured as "sub-tasks" of a single process
- have access to the same memory (shared memory)
- can lead conflicts (improper synchronization)
    - *writing to same memory location at the same time*

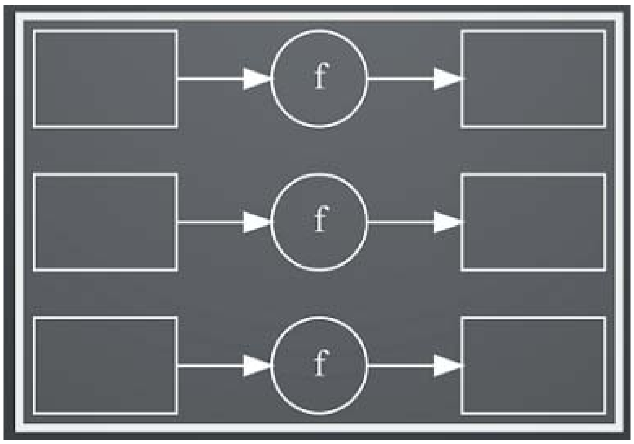
multi-processing

- safer approach (although has communication overhead)
- each process is completed independently from each other

Map function

Used to run a function over multiple elements

```
def square(a):
    return a*a
outputs =[]
for i in inputs:
    outpus.append(square(i))
# or
outputs = [square(i) for i in inputs]
#or
outputs = map(f, inputs)
```



Parallel frameworks

- multiprocessing (https://docs.python.org/2/library/multiprocessing.html)
- ***concurrent.futures*** (https://docs.python.org/3/library/concurrent.futures.html)
- ***joblib*** (https://pythonhosted.org/joblib/)
- ipyparallel (https://ipyparallel.readthedocs.io/en/latest/)
- ***MPI4py*** (http://mpi4py.readthedocs.io/en/stable/)
- Dask (https://dask.pydata.org/en/latest/)

futures (concurrent.futures)

- part of standard library (python 3.2)
- abstract layer on top of Python's threading and multiprocessing modules
- **`executor`**
    - abstract class (can not be used directly)
    - *`ThreadPoolExecutor`* :- multithreading
    - *`ProcessPoolExecutor`* :- multiprocessing
    - submit multiple tasks to `Pool`
    - `Pool` assign tasks and schedule them to run

futures: sum of all primes below n

```
import concurrent.futures
import time

def is_prime(num):
    if num <= 1:
        return False
    elif num <= 3:
        return True
    elif num%2 == 0 or num%3 == 0:
        return False
    i = 5
    while i*i <= num:
        if num%i == 0 or num%(i+2) == 0:
            return False
        i += 6
    return True


def find_sum(num):
    sum_of_primes = 0
    ix = 2
    while ix <= num:
        if is_prime(ix):
            sum_of_primes += ix
        ix += 1
    return sum_of_primes
```

multi threading

```
def sum_primes_thread(nums):
    with concurrent.futures.ThreadPoolExecutor(max_workers = 4) as executor:
        for number, sum_res in zip(nums, executor.map(find_sum, nums)):
            print("{} : Sum = {}".format(number, sum_res))
```

multiprocessing

```
def sum_primes_process(nums):
    with concurrent.futures.ProcessPoolExecutor(max_workers = 4) as executor:
        for number, sum_res in zip(nums, executor.map(find_sum, nums)):
            print("{} : Sum = {}".format(number, sum_res))

if __name__ == '__main__':
    nums = [100000, 200000, 300000]
    start = time.time()
    sum_primes_thread(nums)
    sum_primes_process(nums
    print("Time taken = {0:.5f}".format(time.time() - start))
```

Output when executing `sum_primes_process`

```
100000 : Sum = 454396537
200000 : Sum = 1709600813
300000 : Sum = 3709507114
Time Taken = 0.71783
```

Output when executing `sum_primes_thread`

```
100000 : Sum = 454396537
200000 : Sum = 1709600813
300000 : Sum = 3709507114
Time Taken = 1.2338
```

as_completed & wait

as_completed()

- yeilds results as soon as futures start resolving
- vs `map()` : returns the results in order

wait()

- returns tuple with two sets
- one with completed and other conatins the uncompleted one's

as_completed & wait

```
from concurrent.futures import ThreadPoolExecutor, wait, as_completed
from time import sleep
from random import randint

def return_after_5_secs(num):
    sleep(randint(1, 5))
    return "Return of {}".format(num)

pool = ThreadPoolExecutor(5)
futures = []
for x in range(5):
    futures.append(pool.submit(return_after_5_secs, x))
```

as_completed

```
for x in as_completed(futures):
    print(x.result())
```

wait

```
print(wait(futures))
```

- `wait` controls : `return_when` : `FIRST_COMPLETED, FIRST_EXCEPTION`, `ALL_COMPLETED`

joblib

- another parallel processing library
- developed by authors who work on *scikit-learn*
- also built on top of multiprocessing, multithreading
- ability to use a pool of worker like a context manager, reused across several tasks to be parallized
- if `njobs` set to 1, then it is puerly sequential mode, no overhead of setting up a pool

In [ ]:

```
from joblib import Parallel, delayed
from math import sqrt
Parallel(n_jobs=1)(delayed(sqrt) (i**2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

MPI4py

- python binding for MPI (Message Passing Interface)
- *distributed* parallel programming in python
- Based ob MPI-2 C++ bindings
- Almost all MPI calls supported
- API docs: http://pythonhosted.org/mpi4py/apiref/index.html (http://pythonhosted.org/mpi4py/apiref/index.html)

Minimal mpi4py example

```
from mpi4py import MPI
com = MPI.COMM_WORLD
print("%d of %d" %(comm.Get_rank(), comm.Get_size()))
```

Use **mpirun** and **python** to execute this script

```
$ mpirun -n 4 python script.py
```

Notes:

- MPI_Init is called when mpi4py is imported
- MPI_Finalize is called when the scipt exits

P2P communication

send() and recv()

- one to one - one node to another.
- one to many - one node to all nodes or many of them.
- many to one - many nodes, or all nodes, to one node (usually the master).

```python
from mpi4py import MPI
import time

comm = MPI.COMM_WORLD

rank = comm.rank
size = comm.size
name = MPI.Get_processor_name()

shared = (rank+1)*5

if rank == 0:
    data = shared
    comm.send(data, dest=1)
    comm.send(data, dest=2)
    print 'From rank',name,'we sent',data
    time.sleep(5)

elif rank == 1:
    data = comm.recv(source=0)
    print 'on node',name, 'we received:',data


elif rank == 2:
    data = comm.recv(source=0)
    print 'on node',name, 'we received:',data
```

Collective communication: Bcast

Broadcasting data to all the nodes

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.rank

if rank == 0:
    data = {'a':1,'b':2,'c':3}
else:
    data = None

data = comm.bcast(data, root=0)
print 'rank',rank,data
```

All the nodes have the same values for `data`

```
$mpirun -np 5 python bcast.py
rank 0 {'a':1,'b':2,'c':3}
rank 4 {'a':1,'b':2,'c':3}
rank 1 {'a':1,'b':2,'c':3}
rank 3 {'a':1,'b':2,'c':3}
rank 2 {'a':1,'b':2,'c':3}
```

Collective communication: Scatter

scatter the data elements around the processing nodes

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
   data = [(x+1)**x for x in range(size)]
   print 'we will be scattering:',data
else:
   data = None

data = comm.scatter(data, root=0)
print 'rank',rank,'has data:',data
```

Now we see elements of data is scattered among processors

```
$mpirun -np 5 python scatter.py
we will be scattering: [1, 2, 9, 64, 625]
rank 0 has data: 1
rank 1 has data: 2
rank 3 has data: 9
rank 4 has data: 64
rank 5 has data: 625
```

Note: can only scatter as many elements as you have processors, An error is raised, if you attempt to scatter more elements than your processors

Collectives comm: Gather

Opposite to Scatter, gathers all elements from the worker nodes on master node

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
   data = [(x+1)**x for x in range(size)]
   print 'we will be scattering:',data
else:
   data = None

data = comm.scatter(data, root=0)
data += 1
print 'rank',rank,'has data:',data

newData = comm.gather(data,root=0)

if rank == 0:
   print 'master:',newData
```

Output

```
we will be scattering: [1, 2, 9, 64, 625]
rank 0 has data: 1
rank 2 has data: 9
rank 4 has data: 625
rank 3 has data: 64
rank 1 has data: 2
master collected: [2, 3, 10, 65, 626]
```

MPI4py communications

P2 comm:

- Send(data, dest, tag)
- Recv(data, source, tag)
- send/recv : general Python objects, **slow**
- Send/Recv : continuous arrays, **fast**

Collectives:

- Bcast (Broadcast)
- Scatter
- Gather
- Reduction

Tutorial: http://mpi4py.readthedocs.io/en/stable/tutorial.html (http://mpi4py.readthedocs.io/en/stable/tutorial.html)

MPI4py API reference http://mpi4py.scipy.org/docs/apiref/frames.html (http://mpi4py.scipy.org/docs/apiref/frames.html)

parallelism norms

multi-process, not multi-thread

multi-node, not multi-core

message-passing, not shared memory

Frameworks

- futures/joblib
- dask (data intensive tasks)
  - computer/multicore/node/cluster

on HPC

Normal Python: Single processor

parallel8/12/24 (Single Node)

- python cannot make use of thos 8/12/24 processors
- *mulitprocessing, futures, joblib*
- #PBS -l select=**1**:ncpus=24:mpiprocs=24:mem=160GB

parallel8/12/24 (Multi Node)

- distributed parallelism
- *MPI4py, Dask*
- #PBS -l select=**2**:ncpus=24:mpiprocs=24:mem=160GB

new on the plate

- Bioinformatics services on HPC
  - nextflow pipelines (RNAseq/other NGS)
  - Database services

- cloud services
  - AWS
  - Any Deep learning/ Other GPU jobs

References

- http://sebastianraschka.com/Articles/2014_multiprocessing.html (http://sebastianraschka.com/Articles/2014_multiprocessing.html)
- http://masnun.com/2016/03/29/python-a-quick-introduction-to-the-concurrent-futures-module.html (http://masnun.com/2016/03/29/python-a-quick-introduction-to-the-concurrent-futures-module.html)
- http://pydata.github.io (http://pydata.github.io)

THANKS!
Questions?