Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle Since 1965

Multiprocessor Python

John Feo

Topics

- Threads
- Processes
- Queues
- Pools and Maps
- Array, Values, and Locks
- Managers

Python threads

- Threads are schedulable units of computations executing in the same memory space
- Python threads are easy to use and can share data, but Python's Global Interpreter Lock (GIL) guarantees that ONLY ONE thread runs at a time
- An executing thread gives up control whenever it executes an I/O command, thus

Python threads overlap ONLY computations with I/O

Python threads are NOT a parallel computing construct

Using threads to hide I/O


```
import threading
… objects declared here can be accessed by thread func

def thread_func(n):
  … read some input …
  … do some work …
  … print some results …

for i in range(10):
  threading.Thread(target=thread_func, args=(i,)).start()

print("All done!")
```

10 threads

schedule thread

function to call

function arguments – **LIST**

- What's the first thing printed?
- In what order will the reads and writes of the thread function occur?

Joining threads


```
import threading
… objects declared here can be accessed by thread func

def thread_func(n):
  … read some input …
  … do some work …
  … print some results …

threads = []
for i in range(10):
  t = threading.Thread(target=thread_func, args=(i,))
  threads.append(t)
  t.start()

for t in threads:
  t.join()

print("All done!")
```

create list for threads

append thread to list

wait for threads to finish

Spawning concurrent processes (almost the same)


```
import multiprocessing
… objects declared here can NOT be accessed by process func

def proc_func(n):
  … read some input …
  … do some work …
  … print some results …

processes = []
for i in range(10):
  p = multiprocessing.Process(target=proc_func, args=(i,))
  processes.append(t)
  p.start()

for p in processes:
  p.join()

print("All done!")
```

create list for processes

append process to list

wait for processes to finish

Multiprocessing queues

- Use queues to pass data from processes to master

  multiprocessing.queue – put, get, empty, etc.

- Multiprocessing queues are FIFO and threadsafe
- Queues can handle basically any data type
- *Python hides master to client communication*

Using queues to pass data


```
import multiprocessing
from multiprocessing import Queue

q = Queue()

def proc_func(n):
  … do some work …
  … put results on queue

processes = []
for i in range(10):
  p = multiprocessing.Process(target=proc_func, args=(i,))
  processes.append(t)
  p.start()

for p in processes:
  p.join()

while not q.empty():
  q.get()
```

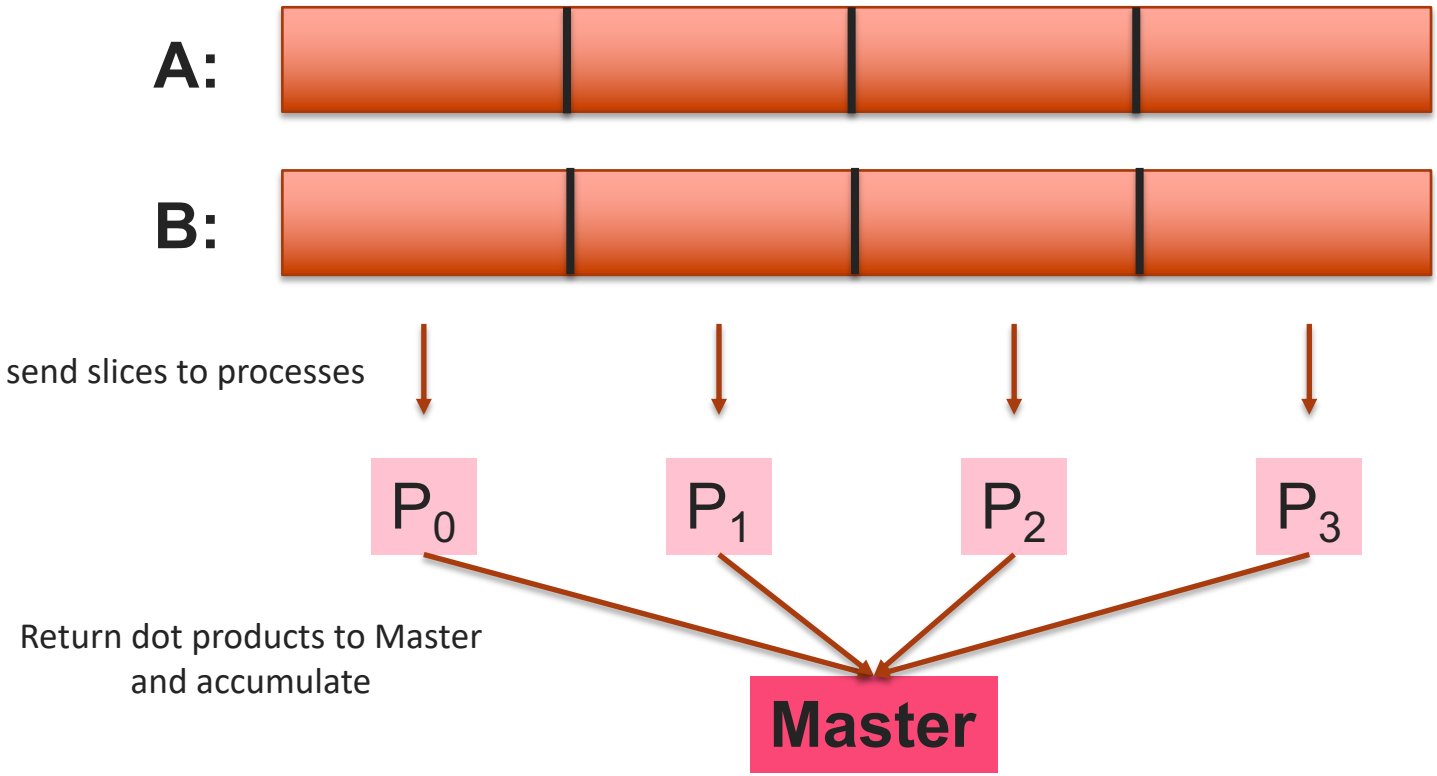
create queue to return data to master

put data on queue

wait for processes to finish

get data from queue

Multiprocessor dot product --- not fast, but simple



Multiprocessor dot product --- setup


```
import os
import sys
import numpy
import multiprocessing
from multiprocessing import Queue

q = Queue()
processes = []

size = 2000000
number_processes = 4
chunk_size = size / number_processes

A = numpy.full(size, 1.0, dtype = numpy.float64)
B = numpy.full(size, 1.0, dtype = numpy.float64)

def proc_func(a, b):
  c = numpy.dot(a, b)
  q.put(c)
```

partition data across processes

compute dot product of my slice
”pass” result back to master

Multiprocessor dot product --- master code


```
for i in range(number_processes):
  lb = i * chunk_size
  up = lb + chunk_size
  p = multiprocessing.Process(target = dotproduct_mp, \
                              args = (a[lb:ub], b[lb:ub],))
  processes.append(p)
  p.start()

for p in processes:
  p.join()

dot = 0.0
while not q.empty():
  dot += q.get()

print("dot product = ", dot)
print("All done")
```

compute range for i^{th} process

slice arrays across processes

aggregate partial results

Log into your account
Review and run dotproduct.mp.py

Multiprocessor pool

- Data-parallel construct similar to Map-Reduce

```
from multiprocessing import pool
pool = Pool(number_cores)
pool.map(func, iterator)
```

- Pools must be closed

```
pool.close()
```

- Pools must be joined (asynchronous)

```
pool.join()
```

Map example


```
import os
import sys
import numpy
import multiprocessing
from multiprocessing import Pool

a = numpy.full(2000000, 1.0, dtype = numpy.float64)
b = numpy.full(2000000, 1.0, dtype = numpy.float64)

def vector_square(x):
  return x * x

pool = Pool(4)
c = pool.map(vector_square, a)
pool.close()
pool.join()
```

Map – passing more than one argument


```
import os
import sys
import numpy
import multiprocessing
from multiprocessing import Pool

a = numpy.full(2000000, 1.0, dtype = numpy.float64)
b = numpy.full(2000000, 1.0, dtype = numpy.float64)

def vector_add(slem_list):
  return sum(elem_list)

pool = Pool(4)
c = pool.map(vector_add, zip(a, b))
pool.close()
pool.join()
```

Map – passing more than one argument


```
import os
import sys
import numpy
import multiprocessing
from functool import partial
from multiprocessing import Pool

size = 2000000
a = numpy.full(size, 1.0, dtype = numpy.float64)
b = numpy.full(size, 1.0, dtype = numpy.float64)

def elem_elem_add(c, d, i):
  return c[i] + d[i]

pool = Pool(4)
map_func = partial(elem_elem_add, a, b)
c = pool.map(map_func, iter(range(size)))
pool.close()
pool.join()
```

create function of N – 1 parameters

must be iterator

Map overhead

- Map overhead is VERY HIGH as every map function is executed by a process
  - vector_square – 14.5 seconds
  - vector_add – 27.7 seconds
  - elem_elem_add – 11.4 seconds
- Need few function calls; functions with LOTS of work

Map – a few functions with lots of work


```
size = 2000000
number_cores = 4
a = numpy.full(size, 1.0, dtype = numpy.float64)
b = numpy.full(size, 1.0, dtype = numpy.float64)

def THICK_elem_elem_add(a, b, number_cores, i):
  slice_size = a.size / number_cores
  lb = slice_size * i;
  ub = lb + slice_size;
  return a[lb:ub] + b[lb:ub]

pool = Pool(number_cores)
map_func = partial(THICK_elem_elem_add, a, b, number_cores)
c = pool.map(map_func, iter(range(number_cores)))
pool.close()
pool.join()
```

big slice, lots of work, equal work

one function call per core

- Execution time = 0.57 seconds (vs. 11.4 seconds)

Log into your account
Review and run pool.py

Multiprocessor value and array

- Value and Array let you share scalar values and arrays across processes

```
from multiprocessing import Value, Array
a = Array('d', [1.0] * size)
b = Array('d', [1.0] * size)
dot = Value('d', 0.0)
```

- Value and Array take 2 parameters
  - type
  - value

Value and array example --- setup code


```
import os
import sys
import multiprocessing
from multiprocessing import Process, Value, Array

size = 2000000
number_processes = 8
chuck_size = size / number_processes

a = Array('d', [1.0] * size)
b = Array('d', [1.0] * size)

dot = Value('d', 0.0)
processes = [ ]

def dotproduct(dot, a, b):
  dot.value += sum(x * y for x, y in zip(a, b))
```

declare shared arrays

declare shared value

NO QUEUE

Value and array example --- master code


```
for i in range(number_processes):
    lb = i * chuck_size
    ub = lb + chuck_size
    p = Process(target=dotproduct, args=(dot, a[lb:ub], b[lb:ub],))
    processes.append(p)
    p.start()

for p in processes:
  p.join()
```

- Same process spawn as with queue
- Log into your account and run *value.array.py*
- **Why do you get the wrong answer?**

Value and array and locks

- Import *Lock*
- Declare a lock

  lock = Lock()

- Add *lock* to *dotproduct* and *args* list
- Acquire and release lock to protect value

  lock.acquire()
  lock.release()

- Run script

Multiprocessor manager

- Manager lets you share more than just scalar values and arrays

```
from multiprocessing import Manager
manager = Manager()
```

- Manager lets you share
  - List
  - Dict
  - Namespace
  - Lock
  - Rlock
  - Semaphore
  - BoundedSemaphore
  - Condition
  - Event
  - Queue
  - Value
  - Array

Bag_of_Words --- setup code


```
import os
import sys
from collections import defaultdict
from multiprocessing import Process, Lock, Manager


file = open('article', mode = 'r')
lines = file.read().split('.')
file.close()

processes = [ ]
manager = Manager()

bag = manager.dict()
lock = manager.Lock()
delims = manager.list([',', ';', ':', '-', '.'])
```

define process queue and manager

define shared values

Bag_of_Words --- function code


```
def bag_of_words(line, l, d, delims):
  my_dict = defaultdict(int)

  for d in delims:
    line = line.replace(d, ' ')

  for word in line.split():
    my_dict[word] += 1

  lock.acquire()

  for key, value in my_dict.iteritems():
    try:
      bag[key] += value
    except KeyError:
      bag[key] = value

  lock.release()
```

local dictionary

remove punctuations

bag words

acquire lock

add words to global bag

release lock

Bag_of_Words --- master code


```
for line in lines:
    p = Process(target=bag_of_words, args=(line, lock, bag, delims,))
    processes.append(p)
    p.start()

for p in processes:
  p.join()
```

- Spawn, start, and join processes as usual
- Note, one process per line, so faster implementations possible
  - How can we spawn fewer processes and do more work per process ??

Log into your account
Review and run manager.py