multiprocessing

HPC Python

R. Todd Evans
rtevans@tacc.utexas.edu

April 17, 2015

What is Multiprocessing

- Process-based parallelism
- Not threading!
- Threads are light-weight execution units within a process that share memory
- Processes are more heavy-weight and do not share memory
- Processes communicate over some interprocess communication channel.

Multiprocessing in Python - Why not threads?

Python Threading Does Not Allow True Concurrency

- Python has a threading module and can use threads
- Python uses a global interpreter lock (GIL)
- In Python, only one thread executes Python code at a time
- GIL avoids concurrent access (race conditions) but no gain in performance for CPU-bound code with multiple threads

Python Multiprocessing Does Allow True Concurrency

- multiprocessing uses subprocesses instead of threads to side-step GIL and allow concurrency in Python code!
- Parent process spawns child processes with own Python interpreter and own GIL
- Each child process inherits the data and program state from parent process

Why Python Multiprocessing

Parallelism!

- Python threads cannot work concurrently due to GIL
- Multiprocessing enables true concurrent (parallel) work across multiple cores
- As modern systems increase core counts, effective utilization of these cores is critical to performance
- If data is restricted to each process significant performance gains are possible
- Spawning processes and sharing data can have significant overhead
- Can scale well to a single node. (16 cores on Stampede & 20 cores on Maverick)
- Python is easy and allows for relatively fast prototyping and high productivity

Multiprocessing Library

- Great implementation of multiprocessing on Python
- multiprocessing provides an interface similar to threading libraries such as OpenMP
- If you know threads, multiprocessing is easy.
- If you don't know threads, multiprocessing is still easy!
- You can communicate Python objects
- What you lose in performance (because Python can be slow), you gain in shorter development time

Quick Example (Gets easier)

```
# multiproc_test.py
import random,os
import multiprocessing

def list_append(count, out_list):
    """
    Appends a
    random number to the list 'count' number
    of times. A CPU-heavy operation!
    """
    print os.getpid(),'is working'
    for i in range(count):
        out_list.append(random.random())

if __name__ == "__main__":
    size = 10000000   # Number of random numbers to add
    procs = 2   # Number of processes to create

    # Create a list of processes and define work for each process
    process_list = []

    for i in range(0, procs):
        out_list = list()
        process = multiprocessing.Process(target=list_append,
                                          args=(size, out_list))
        process_list.append(process)

    # Start the processes (i.e. calculate the random number lists)
    for p in process_list:
        p.start()

    # End all of the processes have finished
    for p in process_list:
        p.join()

    print "List processing complete."
```

Multiprocessing Workflow

Basic Workflow

1. Create Process objects (aka child processes or **worker**) and assign target functions w/ arguments to work on
2. Spawn processes & processes do work
3. Wait for processes to terminate

Basic Syntax

1. `p = Process(target=func, args=(arg0,arg1,...))`
2. `p.start()`
3. `p.join()`

Functionality

Basically same as threading

- Exchange objects/data between processes
  - Queues: multiple producers and consumers of work
  - Pipes: sends data between two processes (`send()`, `recv()`)
- Synchronization - locks etc.
- Sharing data between Processes
  - Shared memory: `Value` (scalar-like) `Array` (vector-like)
  - Server process (Manager): creates and manages shared objects/data
- Pool of Workers: sets up a pool of processes and gives them tasks

Manager + Pool = The safe, easy, and typical approach to multiprocessing. We will focus on this approach.

Process Pools

multiprocessing.pool.Pool(processes): Basic methods

- `apply(func[, args, [kwargs]])`: call func using one worker from Pool. Blocks until complete (not concurrent)
- `apply_async(func[, args[, kwargs]])`: apply that does not block (allows concurrency)
- `map(func, iterable)`: iterable is any object than can be iterated over such as a list or dict. the iterable is divided among workers as chunks. When worker finishes it grabs new chunk. Blocks until complete.
- `map_async(func, iterable)`: map that does not block

apply(): simple example

apply() vs apply_async(): blocking vs non-blocking

```python
# apply_test.py
import time
from multiprocessing import Pool

def f():
    start = time.time()
    time.sleep(2)
    end = time.time()
    return end-start

p = Pool(processes=1)

# apply function

result = p.apply(f) # blocking
print "apply is blocking"
print 'total time',result

# apply_async function
result = p.apply_async(f) # non-blocking
print "apply_async is non-blocking"
while not result.ready():
    time.sleep(0.5)
    print 'working on whatever else I want...'
print 'total time',result.get() # but get() is blocking
```

map(): simple example

```
# map_test.py
import time
from multiprocessing import Pool

def f(x):
    return x**3


y = range(int(1e7))

p = Pool(processes=4)

# map function
start = time.time()
results = p.map(f,y) # blocking
end = time.time()
print "map blocks"
print "time",end-start

# map_async
start = time.time()
results = p.map_async(f,y) # non-blocking
end = time.time()
print "map_async is non-blocking"
output = results.get() # but get() is blocking
print "time",end-start
```

map(): Monte Carlo Integration

```python
# multiproc_mc.py
# integrate f(x) from a to b

from multiprocessing import Pool
from numpy import random
import time,math

def f(x):
    r,s=x
    var = random.random()*(s-r)+r # [0,1] -> [a,b]
    return var**2 # x^2

a = 0 # lower integration bound
b = 1 # upper integration bound
N = 10000000 # number of samples
irange = N*[(a,b)]

# Scalar
random.seed(0)
start = time.time()
samples = map(f,irange) # serial map
end = time.time()
I = (b-a)*sum(samples)/N # compute integral
print 'scalar result',I,end-start

# Multiprocessing
p = Pool(processes=4) # 4 process Pool
random.seed(0)
start = time.time()
samples = p.map(f,irange) # parallel map
end = time.time()
I = (b-a)*sum(samples)/N # compute integral
print 'parallel result',I,end-start
```

`map()`: Prime Factorization

Try with different number of processes N=1,2,3,...

`python multiproc_prime.py N`

```python
# multiproc_prime.py
from multiprocessing import Pool
import time, sys

def is_prime(n):
    if n < 2: return False
    if n < 4: return True
    maxfactors = int(n ** 0.5) + 1
    for i in range(2,maxfactors):
        is_factor = n % i == 0
        if is_factor:
            return False
    return True

numbers_to_test = range(int(1e5))
# Serial
start = time.time()
results = map(is_prime,numbers_to_test)
end = time.time()
print 'Time for serial prime factorization',end-start

# Parallel
processes = int(sys.argv[1])
p = Pool(processes)
# map determines
start = time.time()
results = p.map(is_prime,numbers_to_test)
end = time.time()
print 'Time for parallel prime factorization',end-start
```

Inter-process Data Sharing via `Manager()`

- `map()` map doesn't share data between processes
- Managing shared data can be tricky and error prone.

Let the Multiprocessing Manager do it!

```
manager = Manager()
d = manager.dict()
l = manager.list()
```

`d` and `l` are visible to every process and manager keeps them sync'd

Shared data are seen by all processes

```python
# multiproc_manager.py
from multiprocessing import Manager, Pool
import os

def f(l,d):
    l.append('worker')
    d[str(os.getpid())] = 'worker'
manager = Manager()
pool    = Pool(2)

# private_l and private_d only visible to local process
private_l = list()
private_d = dict()

# shared_l and shared_d visible to every process
shared_l = manager.list()
shared_d = manager.dict()

# manager process can see this change
private_l.append('manager')
private_d[str(os.getpid())] = 'manager'

# manager process can see this change
shared_l.append('manager')
shared_d[str(os.getpid())] = 'manager'

# changes child processes makes are lost
pool.apply(f,args=(private_l,private_d))
pool.apply(f,args=(private_l,private_d))
print "try to add to private data",private_l,private_d

# changes child processes makes are kept
pool.apply(f,args=(shared_l,shared_d))
pool.apply(f,args=(shared_l,shared_d))
print "try to add to shared data",shared_l,shared_d
```

Shared data are seen by all processes

Save those primes!

```
# multiproc_prime_manager.py
from multiprocessing import Pool, Manager
from functools import partial
import time, sys

def shared_is_prime(n,p):
    if n < 2: return False
    if n < 4: return True
    maxfactors = int(n ** 0.5) + 1
    for i in range(2,maxfactors):
        is_factor = n % i == 0
        if is_factor:
            return False
    p.append(n)
    return True

numbers_to_test = range(int(1e5))

# Parallel
processes = int(sys.argv[1])
p = Pool(processes)
manager = Manager()
primes = manager.list()
# map determines
start = time.time()
partial_shared_is_prime = partial(shared_is_prime, p = primes) # trick to pass more than 1 arg
results = p.map(partial_shared_is_prime,numbers_to_test)
end = time.time()
print primes
print 'Time for parallel prime factorization',end-start
```

License