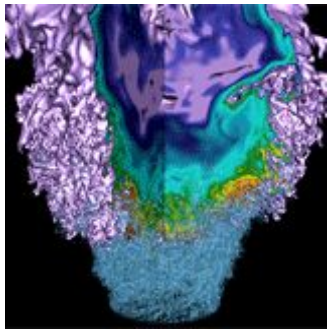
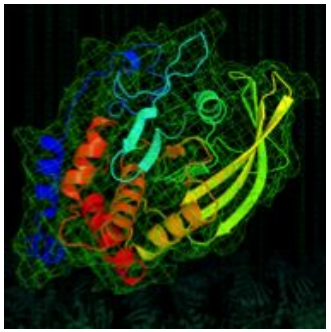
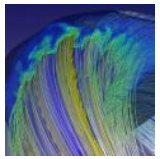
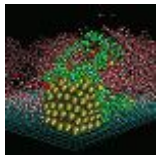
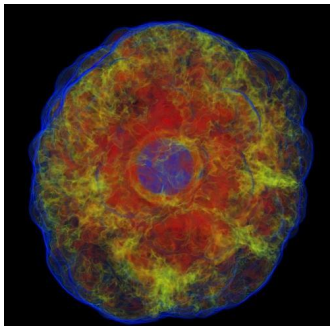
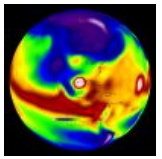
Simple examples of thread-parallel code with python


Debbie Bard

NERSC 40 YEARS at the FOREFRONT 1974-2014

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

Threading in python

- Python runs in single-thread mode
  - "GIL" (Global Interpreter Lock)
- You have to *force* it to run using all threads.
- Two modules for this:
  - *threading* (threads run in same memory space)
  - *multiprocessing* (threads have their own memory space)
- There are obvious implications in memory management here: harder to share objects between processes in *multiprocessing*.
- Big difference in multicore architectures: *multiprocessing* can launch parallel threads on different cores! *threading* can only launch concurrent threads on the *same core*.
- **I prefer *multiprocessing* for this reason.**

Doing a dumb function

Simple function:

```
import math

def my_func(x):
    for i in range(10):
        x+=(math.cos(x) + math.sin(x))
    return x
```

Four different ways of dealing with it: loop/map/numpy

```
from helper import my_func
import sys

n = int(sys.argv[1])

arr = []
for i in range(n):
    arr.append(n)

results = []
for i in range(n):
    results.append(my_func(arr[i]))

print "loop sum:", sum(results1)
```

```
from helper import my_func
import sys

n = int(sys.argv[1])

arr = []
for i in range(n):
    arr.append(n)

results1 = map(my_func, arr)
print "map sum:", sum(results1)
```

```
from numpy import vectorize, arange, sum, shape
from helper import my_func
import sys


n = int(sys.argv[1])

arr = arange(n)
results = vectorize(my_func)
print "numpy.vectorize:", sum(results(arr))
```

multiprocessing

Using multiprocessing:

```
from multiprocessing import cpu_count
from multiprocessing import Pool
from helper import my_func
import sys


n = int(sys.argv[1])

arr = []
for i in range(n):
    arr.append(n)

processes=cpu_count()
print "I have", processes, "cores here"
pool = Pool(processes)


results2 = pool.map(my_func, arr)
pool.close()
pool.join()
print "pool.map:", sum(results2)
```

How many cores?

Instantiate the pool of processes

map the function onto the array, via the pool of processes

Timing tests on Edison @ NERSC

- Timing with 10000-length array on Edison, **10** iterations in the function loop

loop	map	vectorize	multiprocess
real 0m0.119s user 0m0.088s sys 0m0.012s	real 0m0.099s user 0m0.088s sys 0m0.008s	real 0m0.177s user 0m0.148s sys 0m0.024s	real 0m0.144s user 0m0.144s sys 0m0.112s

- How inefficient!
  - There's overhead in instantiating the processes/numpy objects...
- Try a longer array, n=100000:

loop	map	vectorize	multiprocess
real 0m0.778s user 0m0.760s sys 0m0.012s	real 0m0.743s user 0m0.732s sys 0m0.012s	real 0m0.816s user 0m0.756s sys 0m0.056s	real 0m0.205s user 0m1.156s sys 0m0.172s

4x faster than single-thread version!
But each Edison compute node has 24 cores..
Note the CPU time required is ~8 times more than what the user sees - still some overhead.

Timing tests on Edison @ NERSC

- Timing with 10000-length array on Edison, **100** iterations in the function loop

loop	map	vectorize	multiprocess
real 0m0.622s user 0m0.596s sys 0m0.020s	real 0m0.615s user 0m0.600s sys 0m0.012s	real 0m0.713s user 0m0.680s sys 0m0.032s	real 0m0.151s user 0m0.924s sys 0m0.180s

~ same time

- Try a longer array, n=100000, with the 100 iterations:

loop	map	vectorize	multiprocess
real 0m6.130s user 0m6.104s sys 0m0.024s	real 0m5.995s user 0m5.976s sys 0m0.008s	real 0m6.230s user 0m6.180s sys 0m0.048s	real 0m0.552s user 0m9.629s sys 0m0.164s

~12x faster!

Timing tests on my MAcBook Air

- Timing with 10,000-length array on my laptop (4 cores), **100** iterations in the function loop

loop	map	vectorize	multiprocess
real 0m1.250s user 0m1.090s sys 0m0.133s	real 0m1.870s user 0m1.245s sys 0m0.266s	real 0m1.283s user 0m1.081s sys 0m0.143s	real 0m1.153s user 0m1.337s sys 0m0.162s

- Try a longer array, n=100000:

loop	map	vectorize	multiprocess
real 0m4.374s user 0m4.174s sys 0m0.149	real 0m4.424s user 0m4.187s sys 0m0.161s	real 0m4.439s user 0m4.226s sys 0m0.161s	real 0m2.582s user 0m6.411s sys 0m0.174s

1.7x faster!

A 1.7x speedup on my little MacBook Air processor is not too shabby.

What have we learned?

- There is overhead in setting up threads/processes.
- Unless the processes are doing a significant amount of work, it's not worth using multiprocessessing.
- So, when you're thinking about using multi-threading in your code, make sure you split the work appropriately!

A more realistic code example

- Timing test to fit 1D gaussians (using astropy):

```
def my_func2(y):
    x = np.linspace(-5, 5, 100)

    g_init = models.Gaussian1D(amplitude=1., mean=0, stddev=1.)
    fit_g = fitting.LevMarLSQFitter()
    g = fit_g(g_init, x, y)

    return g
```

# Gaussian to fit	loop (loop-fitG.py)	multiprocess (poolmap-fitG.py)
100	real 0m2.573s user 0m2.236s sys 0m0.320s	real 0m4.048s user 0m5.716s sys 0m6.684s
10000	real 1m29.034s user 1m28.582s sys 0m0.436s	real 0m10.593s user 2m33.250s sys 0m6.620s
100000	real 14m58.113s user 14m56.100s sys 0m1.880s	real 1m12.355s user 25m50.773s sys 0m10.145s

>12x faster on Edison (24 cores)

- -> Totally worth it, if your problem is large enough!