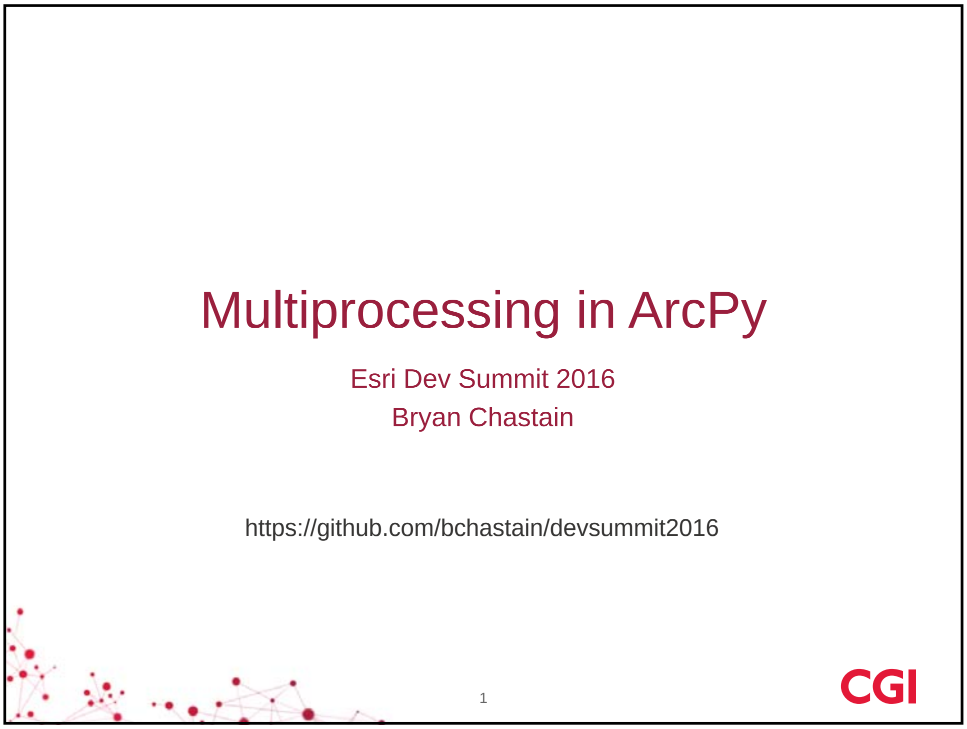
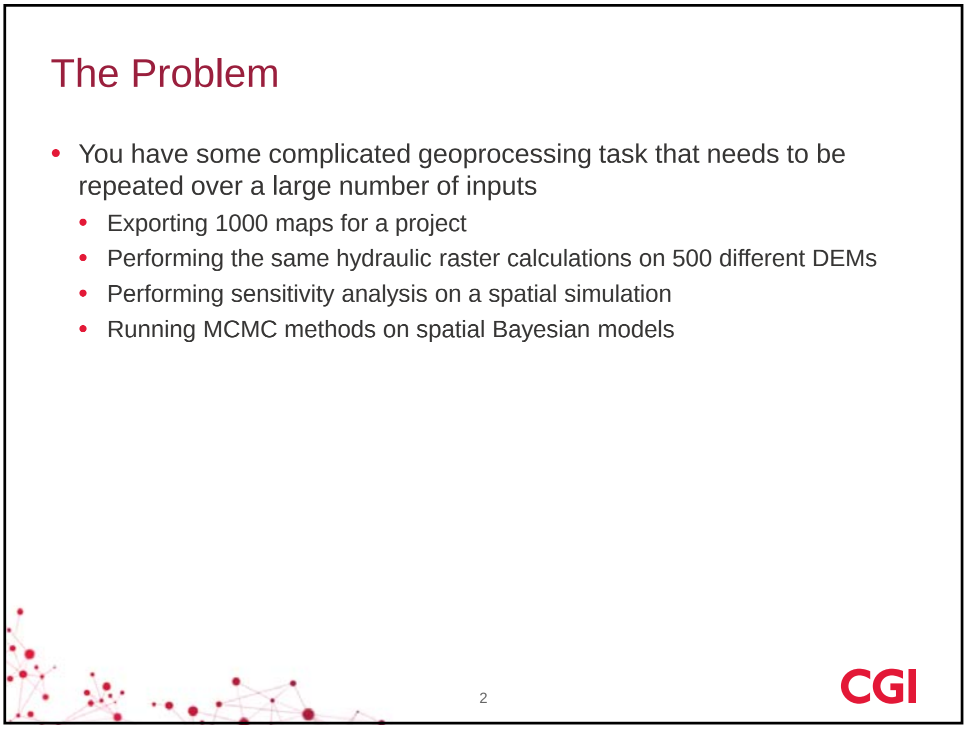


Multiprocessing in ArcPy

Esri Dev Summit 2016
Bryan Chastain

https://github.com/bchastain/devsummit2016



The Problem

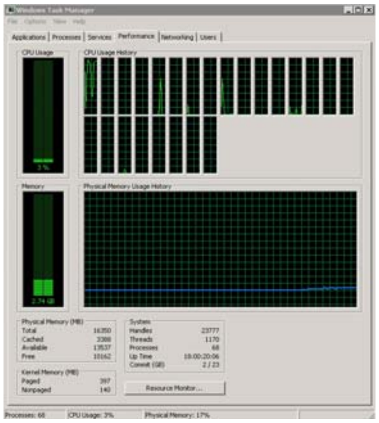
- You have some complicated geoprocessing task that needs to be repeated over a large number of inputs
  - Exporting 1000 maps for a project
  - Performing the same hydraulic raster calculations on 500 different DEMs
  - Performing sensitivity analysis on a spatial simulation
  - Running MCMC methods on spatial Bayesian models

Simple example: export 100 maps

```python
import arcpy

mxd = arcpy.mapping.MapDocument('./testmap.mxd')
out = './images/out'
for i in (0,100):
    arcpy.mapping.ExportToJPEG(mxd, '%s%s.jpg' (out, i), '',
                               1056, 816, 96, '',
                               '8-BIT_GRAYSCALE', 100)
```

CGI

Despite mad Python skills, you still see this:



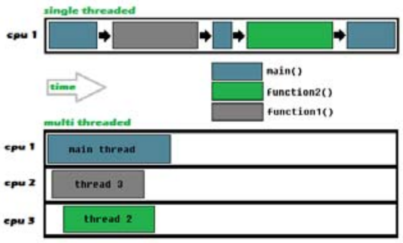
```
Physical Memory (MB)
Total          16350
Cached          3388
Available      13537
Free           10162

Kernel Memory (MB)
Paged            397
Nonpaged         140

System
Handles        23777
Threads         1170
Processes         68
Up Time  18:00:20:06
Commit (GB)   2 / 23

Processes: 68   CPU Usage: 3%   Physical Memory: 17%
```

Script runtime:
375.1s

CGI

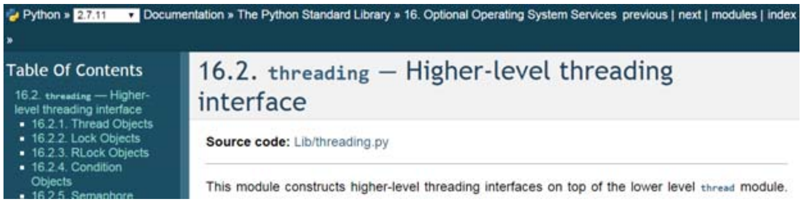
I know what to do!

- Being a smart coder, familiar with similar handling in C++, Java, etc., you think: **Multithreading!**
- Gives ability to use the multiple cores in your machine to concurrently execute multiple tasks in parallel



Hey, cool, a Threading module in Python!

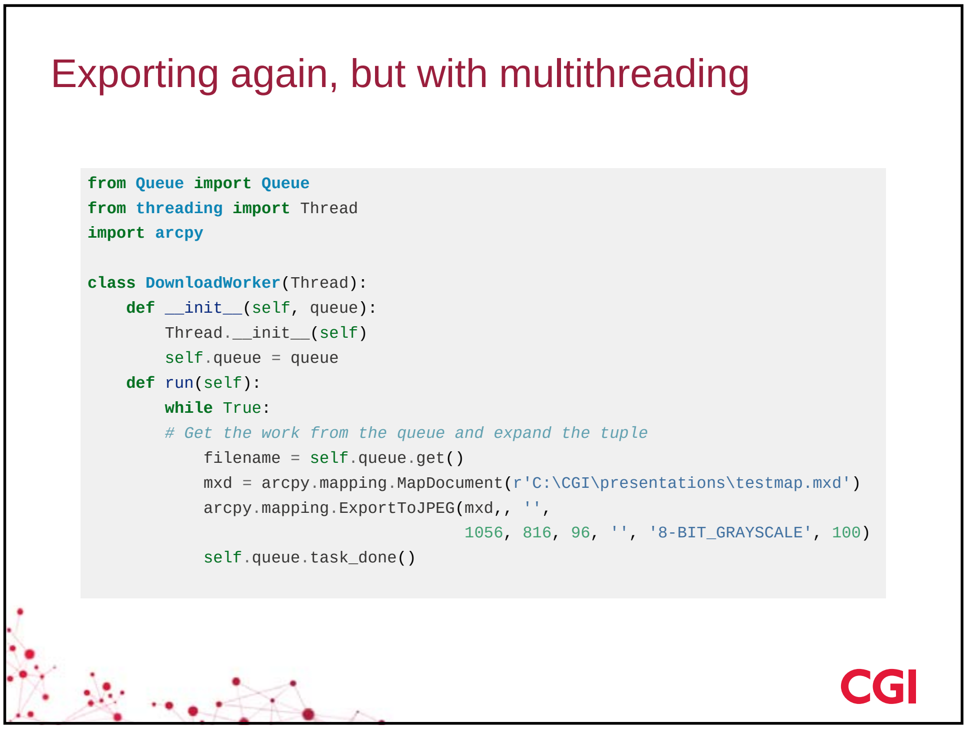
- Easy to use, simply extend the threading.Thread class and override its run() function to set up the code to be parallelized
- The Queue module is also a handy counterpart for delivering work to your new Thread class and flagging it as done


Python » 2.7.11 ▾ Documentation » The Python Standard Library » 16. Optional Operating System Services previous | next | modules | index
»
Table Of Contents
16.2. threading — Higher-level threading interface
16.2.1. Thread Objects
16.2.2. Lock Objects
16.2.3. RLock Objects
16.2.4. Condition Objects
16.2.5. Semaphore

16.2. threading — Higher-level threading interface

Source code: Lib/threading.py

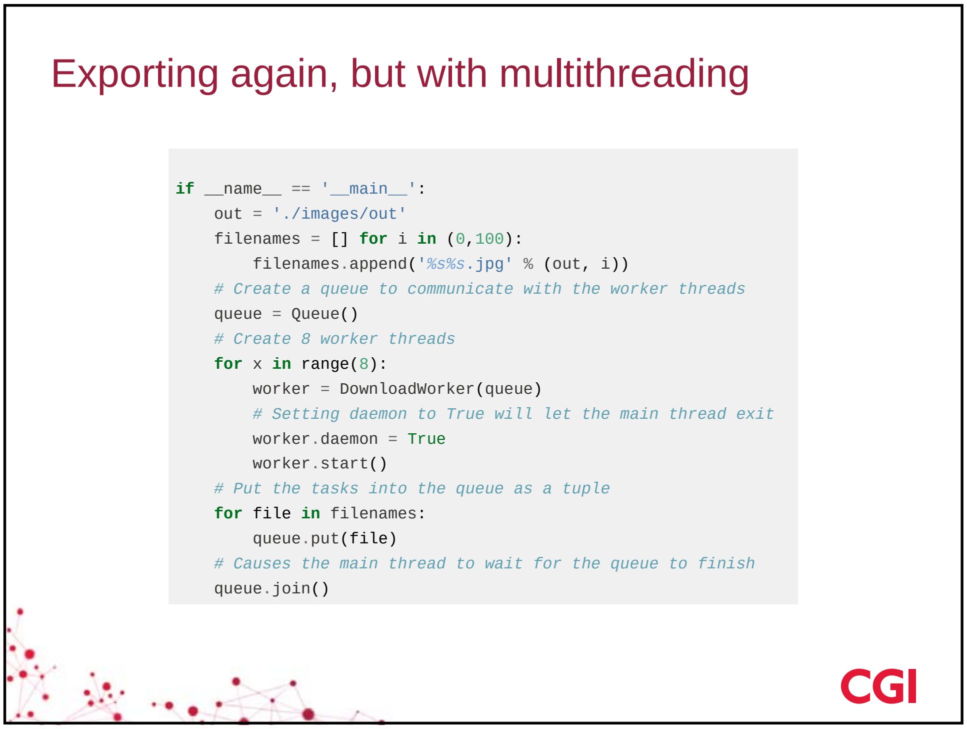
This module constructs higher-level threading interfaces on top of the lower level thread module.



Exporting again, but with multithreading

```
from Queue import Queue
from threading import Thread
import arcpy

class DownloadWorker(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue
    def run(self):
        while True:
            # Get the work from the queue and expand the tuple
                filename = self.queue.get()
                mxd = arcpy.mapping.MapDocument(r'C:\CGI\presentations\testmap.mxd')
                arcpy.mapping.ExportToJPEG(mxd,, '',
                                           1056, 816, 96, '', '8-BIT_GRAYSCALE', 100)
                self.queue.task_done()
```

CGI

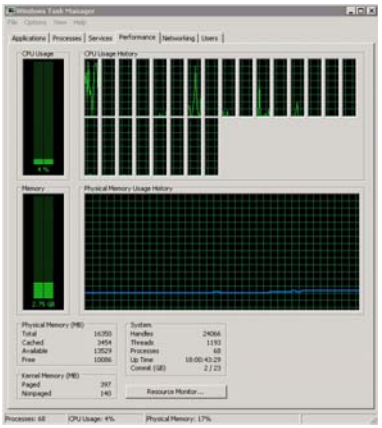


Exporting again, but with multithreading

```
if __name__ == '__main__':
    out = './images/out'
    filenames = [] for i in (0,100):
        filenames.append('%s%s.jpg' % (out, i))
    # Create a queue to communicate with the worker threads
    queue = Queue()
    # Create 8 worker threads
    for x in range(8):
        worker = DownloadWorker(queue)
        # Setting daemon to True will let the main thread exit
        worker.daemon = True
        worker.start()
    # Put the tasks into the queue as a tuple
    for file in filenames:
        queue.put(file)
    # Causes the main thread to wait for the queue to finish
    queue.join()
```

CGI

Python multithreading in action



Physical Memory (MB)	
Total	16350
Cached	3454
Available	13529
Free	10086

Kernel Memory (MB)	
Paged	397
Nonpaged	140

System	
Handles	24066
Threads	1193
Processes	68
Up Time	18:00:43:29
Commit (GB)	2 / 23

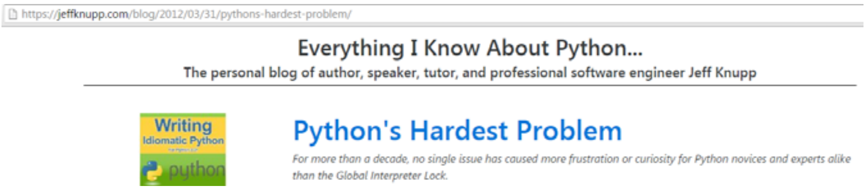
Processes: 68 CPU Usage: 4% Physical Memory: 17%

Script runtime:
713.3s

CGI

Hey wait, what?!

- CPython’s Global Interpreter Lock (GIL)
  - Protects data from being accessed by multiple threads (e.g. current thread state and heap allocated objects for garbage collection)
  - Has the effect of limiting Python to executing one **single thread** at a time
  - For more, read Jeff Knupp’s “Python’s Hardest Problem”



https://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/

Everything I Know About Python...

The personal blog of author, speaker, tutor, and professional software engineer Jeff Knupp

Python's Hardest Problem

For more than a decade, no single issue has caused more frustration or curiosity for Python novices and experts alike than the Global Interpreter Lock.

CGI

Enter the multiprocessing module

- GIL limits us to 1 *thread* at a time, but no limit on *processes*
- Starting in Python 2.6, the multiprocessing module lets you write parallelized code that bypass GIL issues

Python » 2.7.11 ▾ Documentation » The Python Standard Library » 16. Optional Operating System Services previous | next | modules | index
»
Table Of Contents
16.6. multiprocessing — Process-based "threading" interface
▪ 16.6.1. Introduction
▪ 16.6.1.1. The

16.6. `multiprocessing` — Process-based "threading" interface

New in version 2.6.

Usage, in simplest case

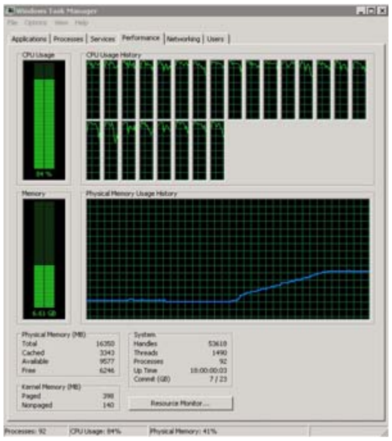
- multithreading.Pool & map()
  - For simple cases, where no synchronization is needed between processes and the tasks are truly independent of one another
- The Pool class represents a pool of worker processes
  - By default is equal to number of CPUs available, but can be set to any number
- pool.map(func, iterable)
  - Parallel equivalent of built-in Python map() function
  - Chops the iterable into a number of chunks which it submits to the process pool as separate tasks

Example

```
import multiprocessing as mp
import time
import arcpy
# Function to map
def exportmap(filename):
    mxd = arcpy.mapping.MapDocument('./testmap.mxd')
    arcpy.mapping.ExportToJPEG(mxd, filename, "", 1056, 816, 96,
                               "", "8-BIT_GRAYSCALE", 100)
if __name__ == '__main__':
    # Optional, default value anyways
    NUM_PROCESSES = mp.cpu_count()
    pool = mp.Pool(NUM_PROCESSES)
    out = './images/out‘
    # Create list of filenames as our iterable
    filenames = ['%s%s.jpg' % (out, i) for i in range(100)]
    pool.map(exportmap, filenames)
```

CGI

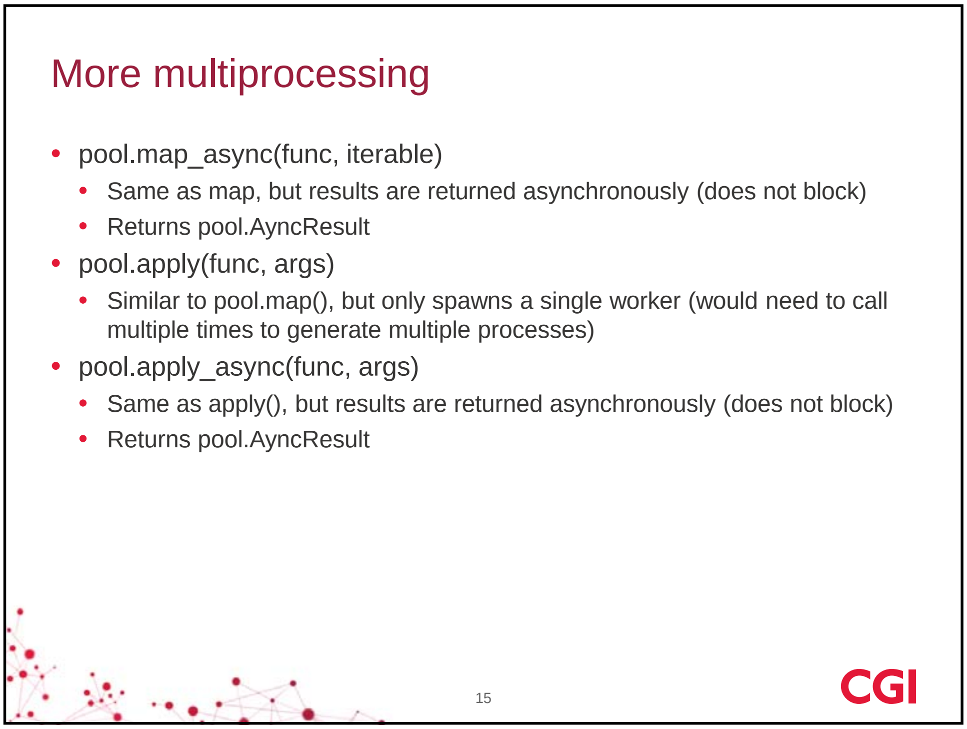
That’s more like it…



```
Physical Memory (MB)
Total        16350
Cached        3343
Available     9577
Free          6246

Kernel Memory (MB)
Paged          398
Nonpaged       140

System
Handles        53610
Threads         1490
Processes         92
Up Time  10:00:00:03
Commit (GB)   7 / 23

Processes: 92    CPU Usage: 84%    Physical Memory: 41%
```

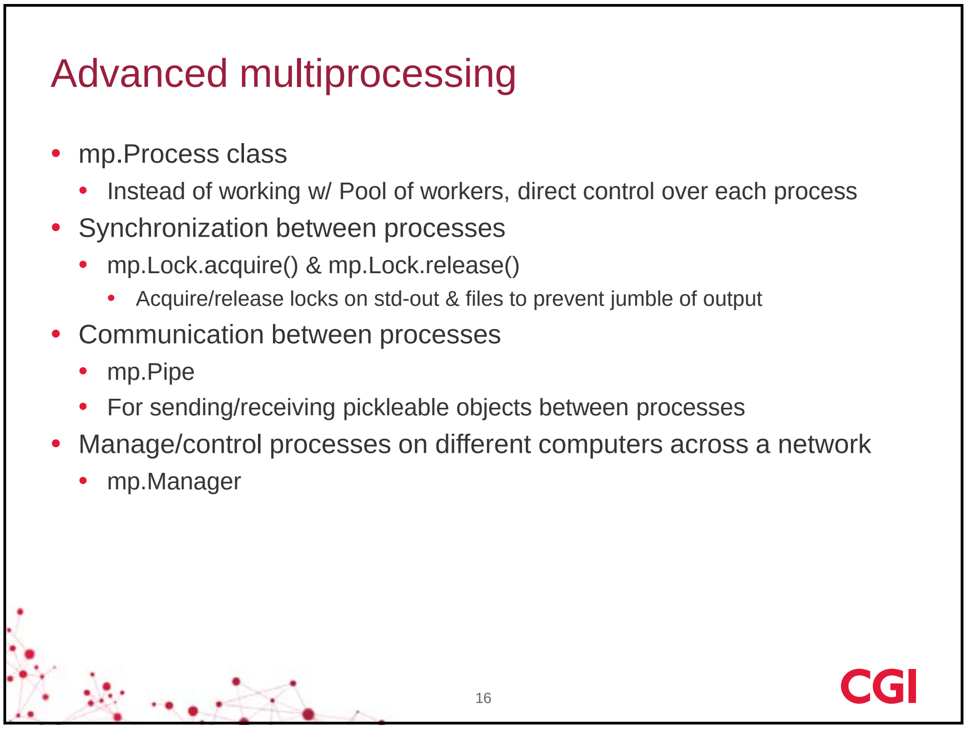
Script runtime:
127.8s

CGI



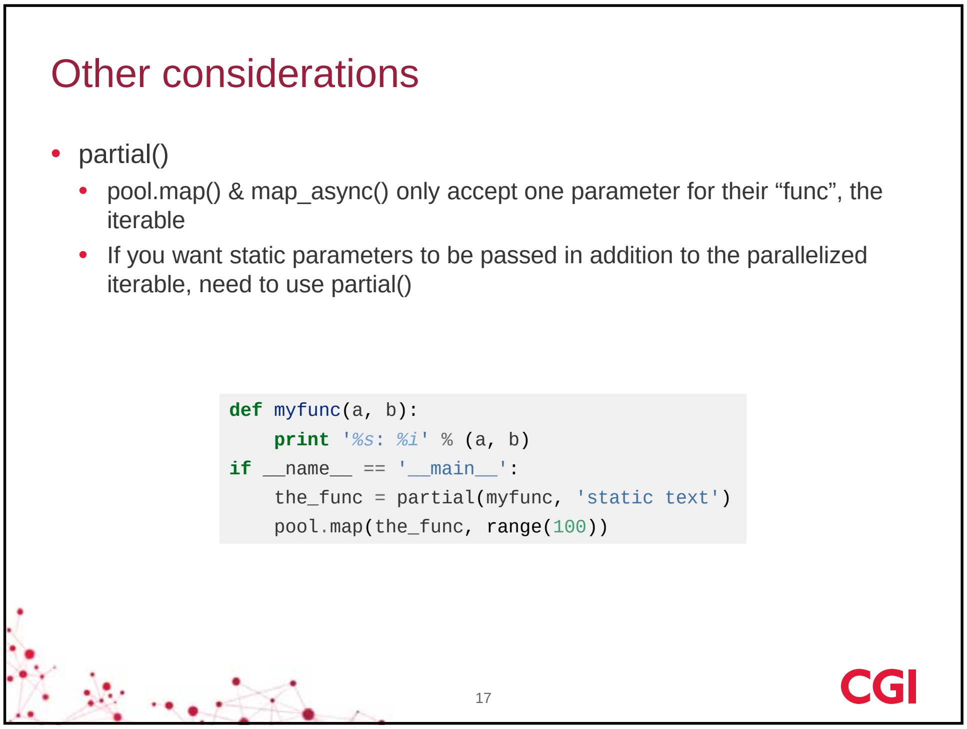
More multiprocessing

- pool.map_async(func, iterable)
  - Same as map, but results are returned asynchronously (does not block)
  - Returns pool.AyncResult
- pool.apply(func, args)
  - Similar to pool.map(), but only spawns a single worker (would need to call multiple times to generate multiple processes)
- pool.apply_async(func, args)
  - Same as apply(), but results are returned asynchronously (does not block)
  - Returns pool.AyncResult



Advanced multiprocessing

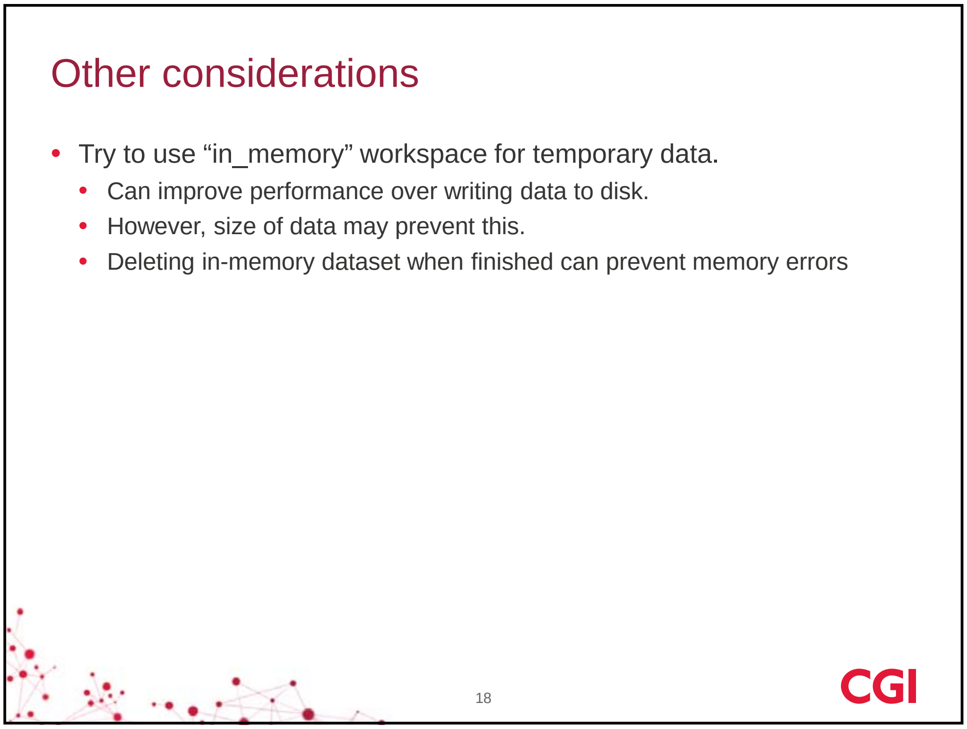
- mp.Process class
  - Instead of working w/ Pool of workers, direct control over each process
- Synchronization between processes
  - mp.Lock.acquire() & mp.Lock.release()
    - Acquire/release locks on std-out & files to prevent jumble of output
- Communication between processes
  - mp.Pipe
  - For sending/receiving pickleable objects between processes
- Manage/control processes on different computers across a network
  - mp.Manager



Other considerations

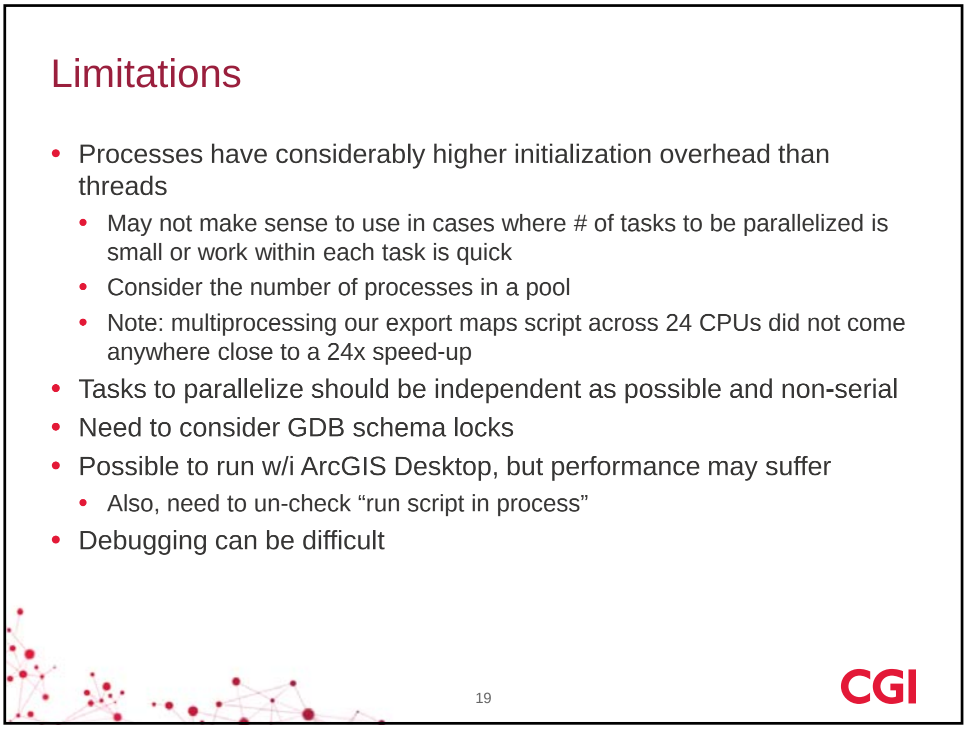
- partial()
  - pool.map() & map_async() only accept one parameter for their “func”, the iterable
  - If you want static parameters to be passed in addition to the parallelized iterable, need to use partial()

```
def myfunc(a, b):
    print '%s: %i' % (a, b)
if __name__ == '__main__':
    the_func = partial(myfunc, 'static text')
    pool.map(the_func, range(100))
```



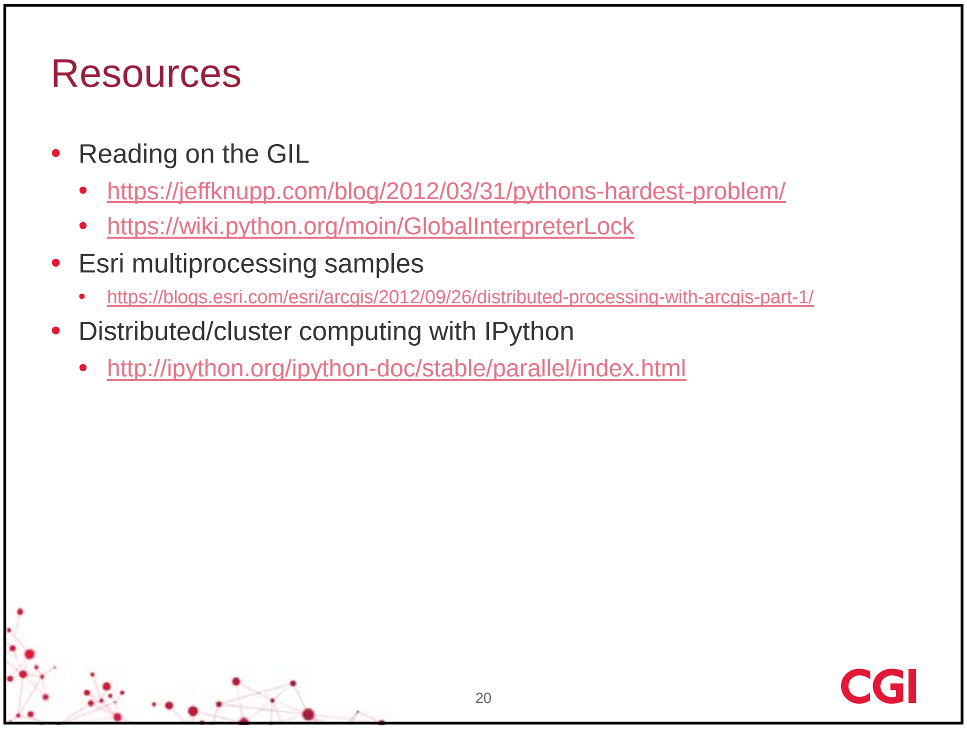
Other considerations

- Try to use “in_memory” workspace for temporary data.
  - Can improve performance over writing data to disk.
  - However, size of data may prevent this.
  - Deleting in-memory dataset when finished can prevent memory errors



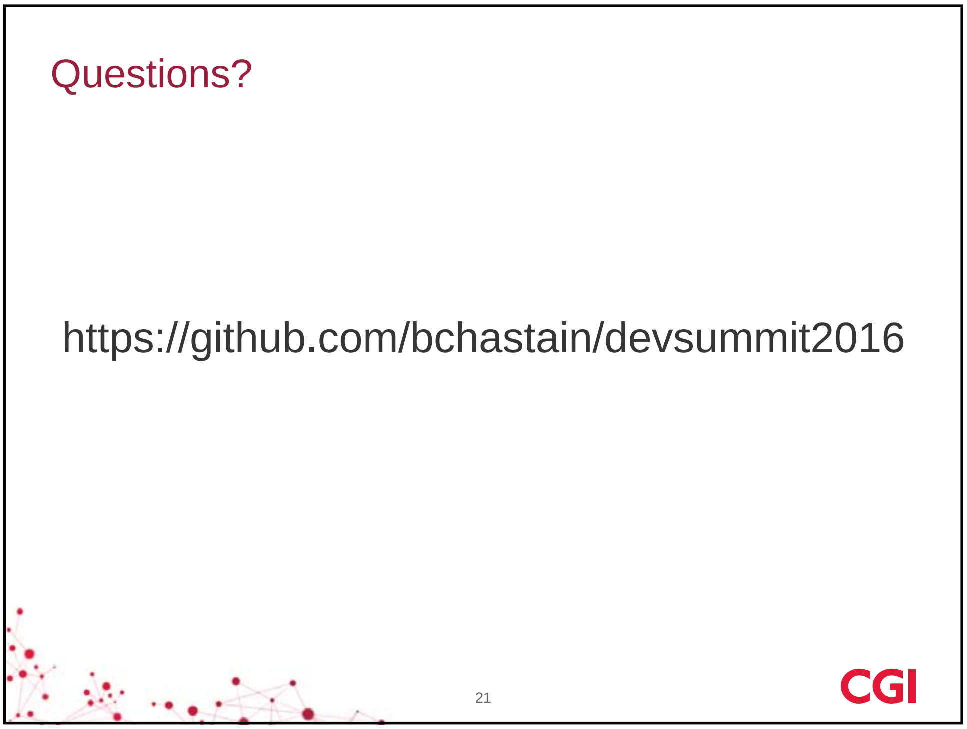
Limitations

- Processes have considerably higher initialization overhead than threads
  - May not make sense to use in cases where # of tasks to be parallelized is small or work within each task is quick
  - Consider the number of processes in a pool
  - Note: multiprocessing our export maps script across 24 CPUs did not come anywhere close to a 24x speed-up
- Tasks to parallelize should be independent as possible and non-serial
- Need to consider GDB schema locks
- Possible to run w/i ArcGIS Desktop, but performance may suffer
  - Also, need to un-check “run script in process”
- Debugging can be difficult



Resources

- Reading on the GIL
  - https://jeffknupp.com/blog/2012/03/31/pythons-hardest-problem/
  - https://wiki.python.org/moin/GlobalInterpreterLock
- Esri multiprocessing samples
  - https://blogs.esri.com/esri/arcgis/2012/09/26/distributed-processing-with-arcgis-part-1/
- Distributed/cluster computing with IPython
  - http://ipython.org/ipython-doc/stable/parallel/index.html



Questions?

https://github.com/bchastain/devsummit2016

CGI