Agenda

- What Multiprocessing Is
- What Multiprocessing Is Not
- Demo of Multiprocessing Modules
  - Multiprocessing
  - Subprocess
  - Asyncio
- Multiprocessing in GIS
  - Raster Concepts
- Demo of Multiprocessing with Rasters
- Vector Concepts
- Demo of Multiprocessing with Vectors

- Best practices and considerations
- Resources
What is Multiprocessing?

- Modern Computers have multiple CPUs
- Single CPU vs. Multiple CPU
- Program ‘Threads’
  - A sewing thread is interwoven fibers
  - Twisted together, tied at the ends
  - A threaded application is interwoven processes
    - Running together, aggregated at the ends
- Single lane road vs. Multi-lane Highway
  - Cost of infrastructure
  - Resilience to traffic jams and accidents
  - Complexity of Interoperability

*Multiprocessing In Python*
What isn’t Multiprocessing?

- **Serial Programs**
  - One Instruction at a time in one process
  - Instruction runs from start to finish
  - Crash ends the whole thing
    - Start over again
    - Fail and close

- **Concurrent Programs**
  - Multiple instructions at a time in one process
  - Instructions may ‘sleep’ to let another run
    - Only one instruction is being executed at any one time

- **Decentralized Programs**
  - Multiple Computers on a network
  - Processes/Instructions run across the network
  - Controlled by a central ‘hub’

- **Distributed Programs**
  - Multiple Computers on a network
  - Processes/Instructions run across the network
  - Each computer operates independently

*Multiprocessing In Python*
Python Multiprocessing Ideals

• Replace all loops with parallel iteration
• Replace all collections with iterators/generators
• Combine Multiprocessing and Concurrency
  - Parallel functions with concurrent instructions
• Fault Tolerance
  - A failed process does not halt the application
  - Ability to ‘try again’ in parallel
• Throttled by input or ‘mapping’ function
  - Validate, send data to available CPUs
• Two forms of output
  - Discrete returns individually processed units of data
  - Aggregated ‘reduces’ combined units of data to a collection
Python Modules

- **threading**
  - Don’t use unless you have a very specific reason to do so
    - core developers
  - Global Interpreter Lock
    - Two threads controlled by a single python.exe cannot run at the same time

- **multiprocessing**
  - Creates multiple python.exe instances
    - Not subject to GIL problem
  - Operating System deals with threading of python.exe

- **subprocess**
  - Use to launch non python.exe processes
    - Serial or Parallel
  - Callback allows subprocess to run in parallel
Python Modules con’t

• twisted
  - ‘Twists’ threads from threading for you
  - Open source Python package
  - Designed to handle I/O concurrency

• asyncio
  - Pure Python Package
  - Designed to handle I/O concurrency
  - Brand new! Fully accepted in Python 3.6.0
Multiprocessing In Action

```python
import multiprocessing
import time

from tqdm import trange

data = enumerate([['a', 2], ['b', 4], ['c', 1], ['d', 0]],
                  i=inputs[0]
                  d, t = inputs[1][i]
for i in trange(20, desc=d, position=i):
time.sleep(t)

def mp_worker(inputs):  
i = inputs[0]
d, t = Inputs[1][i]
for i in range(20, desc=d, position=i):
time.sleep(t)

def mp_handler():
p = multiprocessing.Pool(4)
p.map(mp_worker, data)

if __name__ == '__main__':
    mp_handler()
```
Subprocess In Action

```python
import os
import subprocess
import sys
import time

from tqdm import trange, tqdm

path = os.path.dirname(sys.executable) + os.sep + "Scripts" + os.sep
sys.path.append(path)
exe = "conda.exe"
args = ["list", "--json"]

def sp_handler():
    s = None
    for i in trange(10, desc='Python.exe', position=1):
        if i is not 5:
            time.sleep(1)
        else:
            s = subprocess.check_output([exe, args[i]])
            l = s.split()
            for p in trange(len(l), desc='conda list', position=2):
                tqdm.write(l[p].decode())
            time.sleep(0.1)

if __name__ == '__main__':
    sp_handler()
```

Subprocessing Demo
import os
import subprocess
import sys
import time

from tqdm import tqdm, tqdm_notebook

path = os.path.dirname(sys.executable) + os.sep + "Scripts" + os.sep
sys.path.append(path)
exe = "conda.exe"
args = ["list", " --json"]

def sp_handler():
    s = None
    for i in tqdm(range(10), desc='Python.exe', position=1):
        if i is not s:
            time.sleep(1)
        else:
            s = subprocess.check_output([exe, args[0]])
        l = s.split()
        for p in tqdm([l[1], desc='conda list', position=2]):
            print(l[p], decode())
            time.sleep(0.1)

if __name__ == '__main__':
    sp_handler()
Multiprocessing and Concurrency In GIS
Considerations

- “GIS is the Language of Geography” – Jack Dangermond, Esri UC 2004
- “Python is the Language of GIS” – Bill Moreland, Esri Dev Summit 2014
- GIS is inherently more complex than Python on its own
  - Combining Programming with Geography
- Challenges
  - Projected Data
  - User Interface Considerations
Working with Rasters in Multiprocessing

Candidate operations for parallelization:

- Parallelize independent tasks within a workflow
- Process a large raster parallely (limited to local/focal/zonal operations)
Pleasingly parallel problems

As time proceeds…

Why is multiprocessing relevant to Geoprocessing workflows?

Serialized execution of a model workflow
Pleasingly parallel problems

Parallelized execution of a model workflow

Why is multiprocessing relevant to Geoprocessing workflows?

Worker-1: Slope
Worker-2: Aspect
Worker-3: Reclassify
Worker-4: Weighted Sum

Elevation Raster → Worker-1 → Slope → [Logic] → Worker-4 → Weighted Sum
Elevation Raster → Worker-2 → Aspect → [Logic] → Worker-4 → Weighted Sum
Land use Raster → Worker-3 → Reclassify → [Logic] → Worker-4 → Weighted Sum

Output suitability raster

As time proceeds...
Working with Rasters in Multiprocessing

Candidate operations for parallelization:

- Parallelize independent tasks within a workflow
- Process a large raster parallelly (limited to local/focal/zonal operations)
Easy-to-parallelize raster operations

- Four types of raster operations:
  - Local
  - Focal
  - Zonal
  - Global

Local raster operation

Focal raster operation

Multiprocessing in GIS
Pleasingly parallel problems

Why is multiprocessing relevant to Geoprocessing workflows?

Tool executed serially on a large input dataset
Pleasingly parallel problems

Why is multiprocessing relevant to Geoprocessing workflows?

Tool executed parallelly on a large input dataset
Demo of multiprocessing with Rasters

Run a local raster operation parallelly on a large raster dataset
Demo of multiprocessing with Rasters

```python
# import necessary modules
import arcpy
import multiprocessing
import os
import glob
import sys
import time
import logging
from multiprocessing import Process, Queue, Pool, 
    cpu_count, current_process, Manager

# set global arcpy environments
arcpy.env.overwriteOutput = True
arcpy.CheckOutExtension("Spatial")
arcpy.env.scratchWorkspace = "in_memory"

# create a logger to report
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
formatter = logging.Formatter("%(asctime)s - %(message)s")
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)
ch.setFormatter(formatter)
logger.addHandler(ch)

# describe paths
in_raster_path = os.path.join(os.getcwd(), r'input_raster.tif')
out_fishnet_path = os.path.join(os.getcwd(), r'fishnets', r'fishnet.shp')

def create_fishnet(in_raster_path, out_fc_path):
    ...

def execute_task(in_extentDict):
    ...

if __name__ == '__main__':
    ...
```

Run a local raster operation parallelly on a large raster dataset
Demo of multiprocessing with Rasters

if __name__ == '__main__':
    # start the clock
    time_start = time.clock()
    # call create fishnet functionlarge6insect1kuk
    logger.info("Creating fishnet features...")
    create_fishnet(in_raster_path, out_fishnet_path)
    # get extents of individual features, add it to a dictionary
    extDict = {}
    count = 1
    for row in arcpy.da.SearchCursor(out_fishnet_path, ['SHAPE@']):
        extent_curr = row[0].extent
        ls = []
        ls.append(extent_curr.XMin)
        ls.append(extent_curr.YMin)
        ls.append(extent_curr.XMax)
        ls.append(extent_curr.YMax)
        extDict[count] = ls
        count += 1

    # create a process pool and pass dictionary of extent to execute task
    pool = Pool(processes=cpu_count())
    pool.map(execute_task, extDict.items())
    pool.close()
    pool.join()

    # add results to mosaic dataset
    arcpy.env.workspace = os.getcwd()
    in_path = os.path.join(os.getcwd(), r"local_rast workspace")
    arcpy.AddRastersToMosaicDataset_management("mosaic_out.gdb\localFn_Mosaic", "Raster Dataset", in_path)

    # end the clock
    time_end = time.clock()
    logger.info("Time taken in main in seconds(s) is : {}".format(str(time_end - time_start)))

Run a local raster operation parallely on a large raster dataset
Demo of multiprocessing with Rasters

```python
def create_fishnet(in_raster_path, out_fc_path):
    # create raster object from in_raster_path
    ras1 = arcpy.Raster(in_raster_path)
    # specify input parameters to fishnet tool
    XMin = ras1.extent.XMin
    XMax = ras1.extent.XMax
    YMin = ras1.extent.YMin
    YMax = ras1.extent.YMax
    origCord = "{} {}".format(XMin, YMin)
    YAxisCord = "{} {}".format(XMin, YMax)
    CornerCord = "{} {}".format(XMax, YMax)
    cellSizeW = '0'
    cellSizeH = '0'
    numRows = 4
    numCols = 4
    geo_type = "POLYGON"
    # Run fishnet tool
    logger.info("Running fishnet creator: {} with PID {}").format(current_process().name, os.getpid())
    arcpy.env.outputCoordinateSystem = ras1.spatialReference
    arcpy.CreateFishnet_management(out_fc_path, origCord, YAxisCord, cellSizeW, cellSizeH, numRows, numCols, CornerCord, "NO_LABELS", "", geo_type)
    arcpy.CleanEnvironment("outputCoordinateSystem")
```

Run a local raster operation parallely on a large raster dataset
Demo of multiprocessing with Rasters

```python
def execute_task(in_extentDict):
    # start the clock
    time1 = time.clock()
    # get extent count and extents
    fc_count = in_extentDict[0]
    procExt = in_extentDict[1]
    XMin = procExt[0]
    YMin = procExt[1]
    XMax = procExt[2]
    YMax = procExt[3]
    # set environments
    arcpy.env.snapRaster = in_raster_path
    arcpy.env.cellsize = in_raster_path
    arcpy.env.extent = arcpy.Extent(XMin, YMin, XMax, YMax)
    # send process info to logger
    logger.info("Running local math task: {} with PTD {}".format(current_process().name, os.getpid()))
    # run the local task
    ras_out = arcpy.sa.Sqrt(in_raster_path)
    # clear the extent environment
    arcpy.ClearEnvironment("extent")
    # specify output path and save it
    out_name = "out_sqrt_ras{}.tif".format(fc_count)
    out_path = os.path.join(os.getcwd(), r"local_rast_wspace", out_name)
    ras_out.save(out_path)
    # end the clock
    time2 = time.clock()
    logger.info("{} with PID {} finished in {}".format(current_process().name, os.getpid(), str(time2-time1)))
```

Run a local raster operation parallelly on a large raster dataset
Demo of multiprocessing with Rasters

Run a local raster operation parallelly on a large raster dataset.

Time taken in main in seconds is: 114.30550452605383
Raster analysis performance improvements with multiprocessing

Execute tool in parallel on 57344*57344 cells

Method: Focal function in parallel; 16 chunks created

Run a local raster operation parallelly on a large raster dataset
Raster analysis performance improvements with multiprocessing

Run a local raster operation parallelly on a large raster dataset
Working with Vectors in Multiprocessing

- Vectors are discrete units of data
  - Concept aligns easily with multiprocessing
  - Each unit of data can be mapped to an independent process
  - To aggregate return results:
    - Independent processes return results to a collection
    - Minimal amount of functionality on aggregation step
    - Collection can be returned to calling process or yielded to chained process
    - Example – for each point in a collection, project the point, add to array, return array as line
  - To return discrete results:
    - Independent processes yield results to calling process or chained process
    - Example – for each location-enabled tweet in a collection, geocode the coordinates, return the address
Example of multiprocessing with Vectors

```python
def main():
    """Run Script."""
    ranges = [[0, 250000], [250001, 500000], [500001, 750000], [750001, 1000001]]
    # Create a pool class and run the jobs--the number of jobs is
    # equal to the length of the oids list
    pool = multiprocessing.Pool()
    result_arrays = pool.map(generate_near_table, ranges)

    # Concatenate the resulting arrays and create an output table
    # reporting any identical records.
    result_array = numpy.concatenate(result_arrays, axis=0)
    arcpy.da.NumPyArrayToTable(result_array, 'c:/testing/testing.gdb/gn3')

    # Synchronize the main process with the job processes to
    # Ensure proper cleanup.
    pool.close()
    pool.join()
# End main
```

Vectors in ArcPy
Example of multiprocessing with Vectors

```python
def generate_near_table(ranges):
    """Generate a near table for 300 random points."""
    i, j = ranges[0], ranges[1]
    lyr = arcpy.management.MakeFeatureLayer(
        'c:/testing/testing.gdb/random1mil',
        'layer{}'.format(i), """"OID >= {0} AND
        OID <= {1}"""".format(i, j))
    gn_table = arcpy.analysis.GenerateNearTable(
        lyr, 'c:/testing/testing.gdb/random300',
        'in_memory/outnear{}'.format(i))
    result_array = arcpy.da.TableToNumPyArray(gn_table, ["*"])
    arcpy.management.Delete(gn_table)
    return result_array
    # End generate_near_table function
```
Example of multiprocessing with Vectors

```python
import os
import arcpy

inTable = r"C:\Test\table"

inDesc = arcpy.Describe(inTable)
oldName = arcpy.AddFieldDelimiters(inTable,
inDesc.oidFieldName)

sql = "select min(%) from %" % (oldName,
oldName,
os.path.basename(inTable))
cur = arcpy.da.SearchCursor(inTable,
[inDesc.oidFieldName],
sql)
minOID = cur.next()[0]
del cur, sql

sql = "select max(%) from %" % (oldName,
oldName,
os.path.basename(inTable))
cur = arcpy.da.SearchCursor(inTable,
[inDesc.oidFieldName],
sql)
maxOID = cur.next()[0]
del cur, sql

# 2K slices
breaks = range(minOID, maxOID)[0:1:2000]
breaks.append(maxOID+1)
exprList = [oldName + ' >= ' + str(breaks[b]) + ' and ' +
oldName + ' < ' + str(breaks[b+1]) for b in range(len(breaks)-1)]
```

Vectors in Arcpy
ArcPy Multiprocessing Best Practices

- Use “in_memory” workspace to store temporary results.

- Avoid writing to FGDB data types or GRID raster data types. These data formats can often cause schema lock/synchronization issues.

- Use ArcGIS Pro 1.4 OR ArcGIS Server 10.5 OR ArcMAP with ArcGIS for Desktop-Background Geoprocessing (64-bit). Using 64-bit processing to perform analysis on systems with large amounts of RAM may help when processing large data which may have otherwise failed in a 32-bit environment.
Looking ahead

• Dask – Parallelization on local machine or distributed

• Asynchrony – Utilizing asyncio to concurrently read/write and process data
  - Python 3.6+

• Enhanced Interoperability with Services – Utilizing services is natural with multiprocessing.
  - Deploy as packages or through desktop publishing workflow
  - Chaining Services together through scripts
Resources

- Obtain sample scripts and data that you saw in the demos - https://github.com/nRajasekar92/DevSummit-2017
- Python 3.5 multiprocessing API - https://docs.python.org/3.5/library/multiprocessing.html