Parallel Python: Multiprocessing With ArcPy

Clinton Dow – Geoprocessing

Neeraj Rajasekar – Spatial Analyst

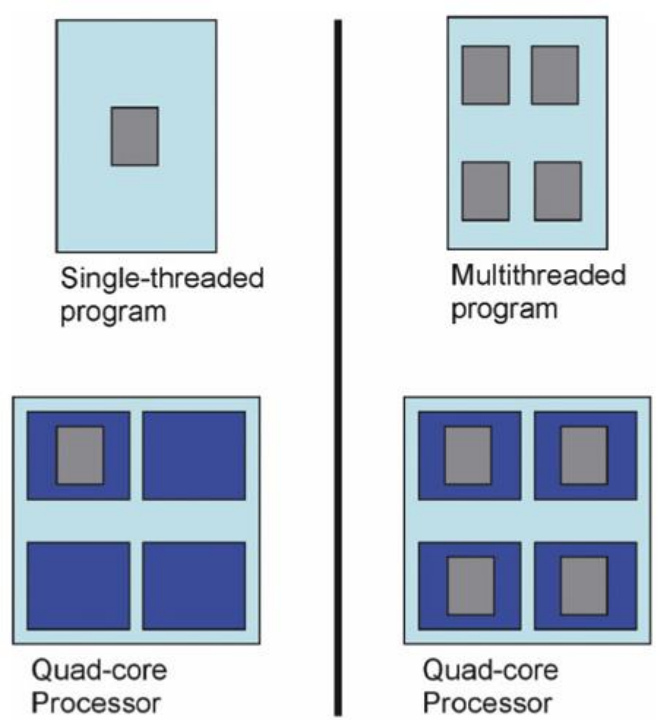
Agenda

- 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

Multiprocessing and Concurrency In Pure Python

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



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 independantly

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

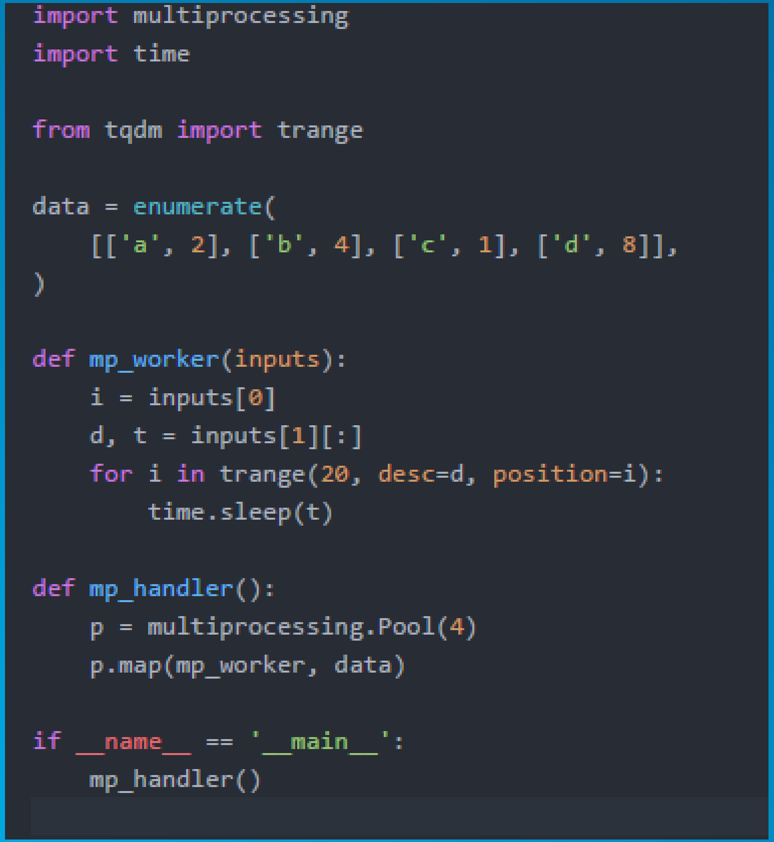
Python Modules

- 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

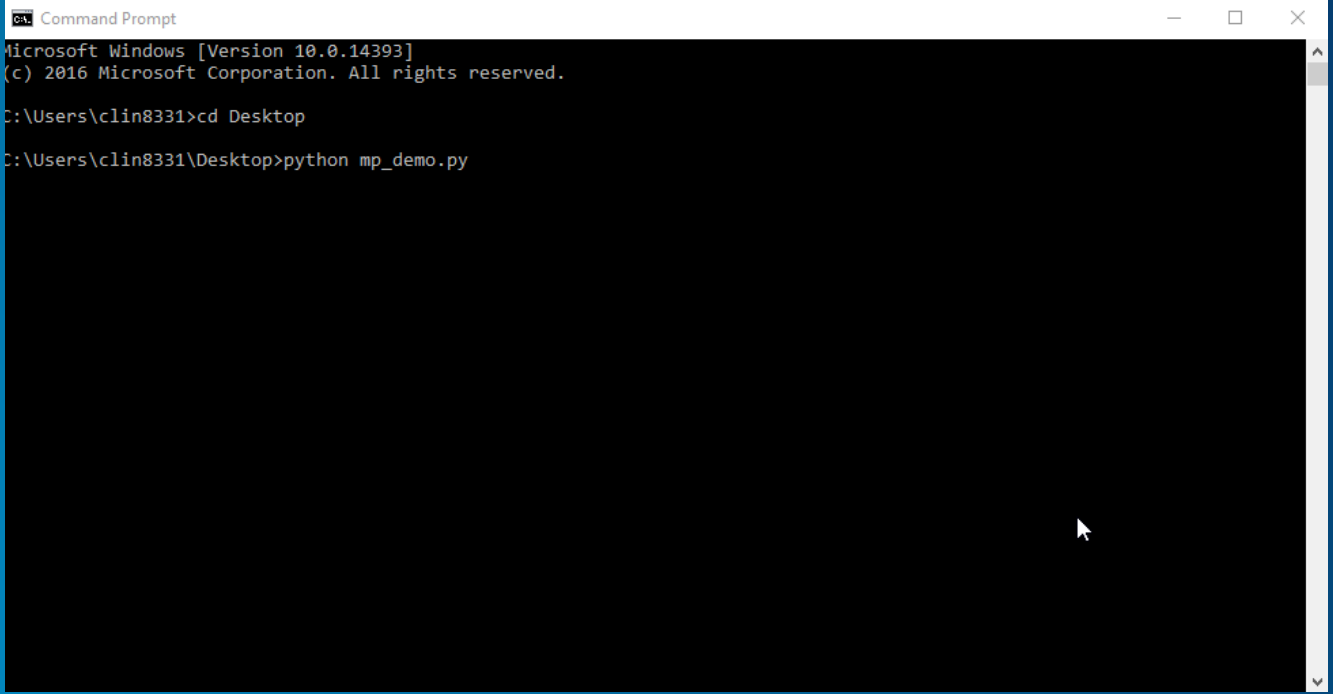
Python Modules con’t

- **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', 8]],
)

def mp_worker(inputs):
    i = inputs[0]
    d, t = inputs[1][:]
    for i in trange(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()
```

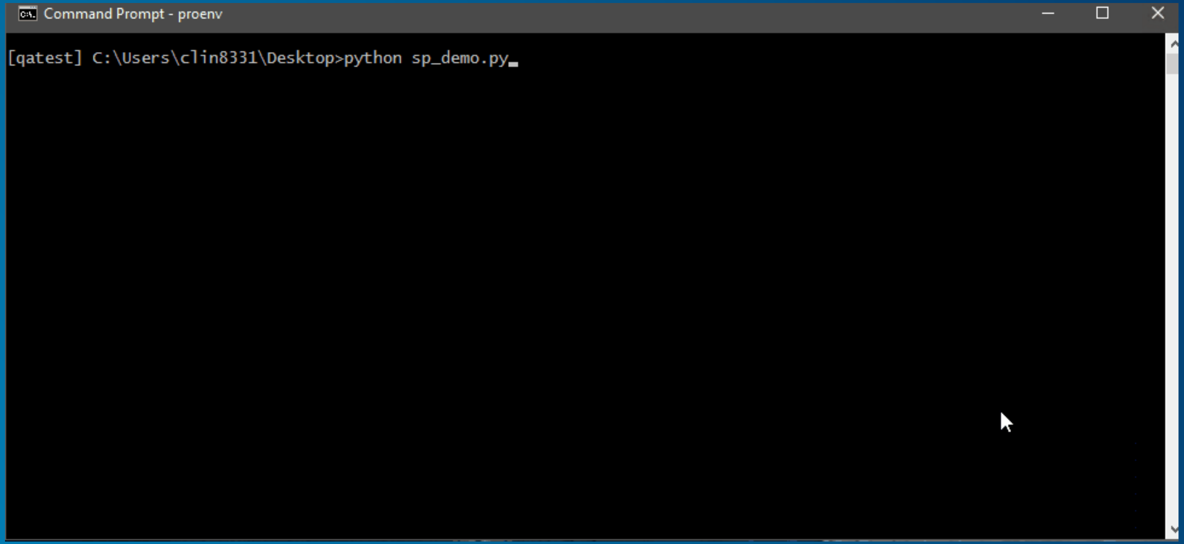


```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\clin8331>cd Desktop

C:\Users\clin8331\Desktop>python mp_demo.py
```

Multiprocessing Demo

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[:]])
            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()
```

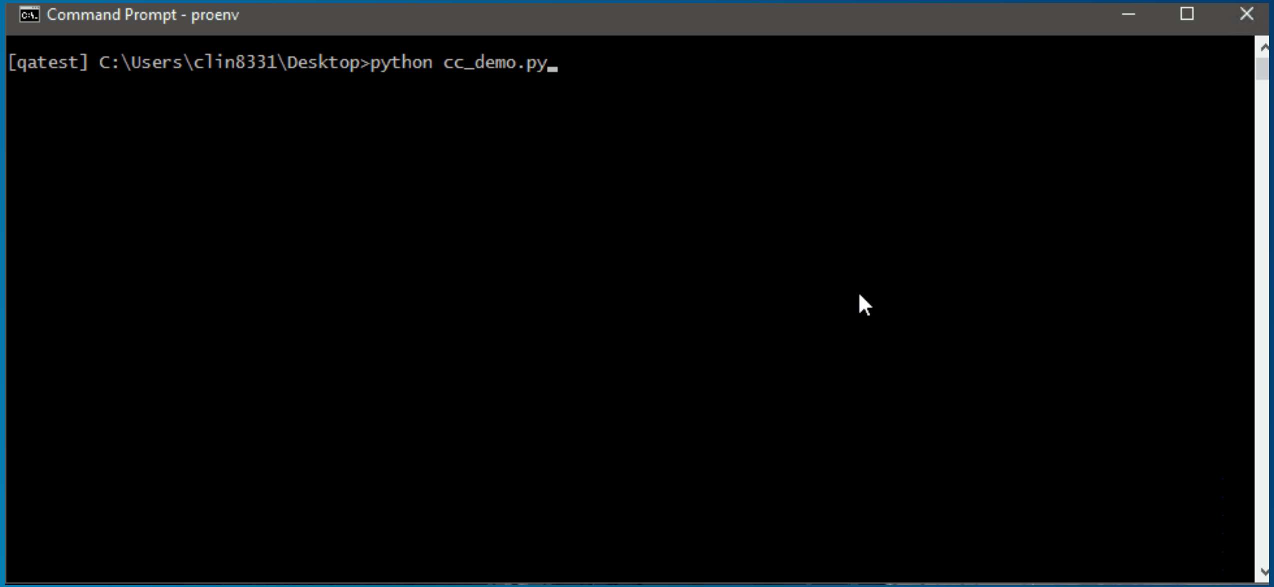


```
[qatest] C:\Users\clin8331\Desktop>python sp_demo.py_
```

Subprocessing Demo

Concurrency 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[:]])
            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()
```



```
[qatest] C:\Users\clin8331\Desktop>python cc_demo.py_
```

Concurrency Demo

Multiprocessing and Concurrency In GIS

Considerations

- **“GIS is the Language of Geography” – Jack Dangermond, Esri UC 2004**

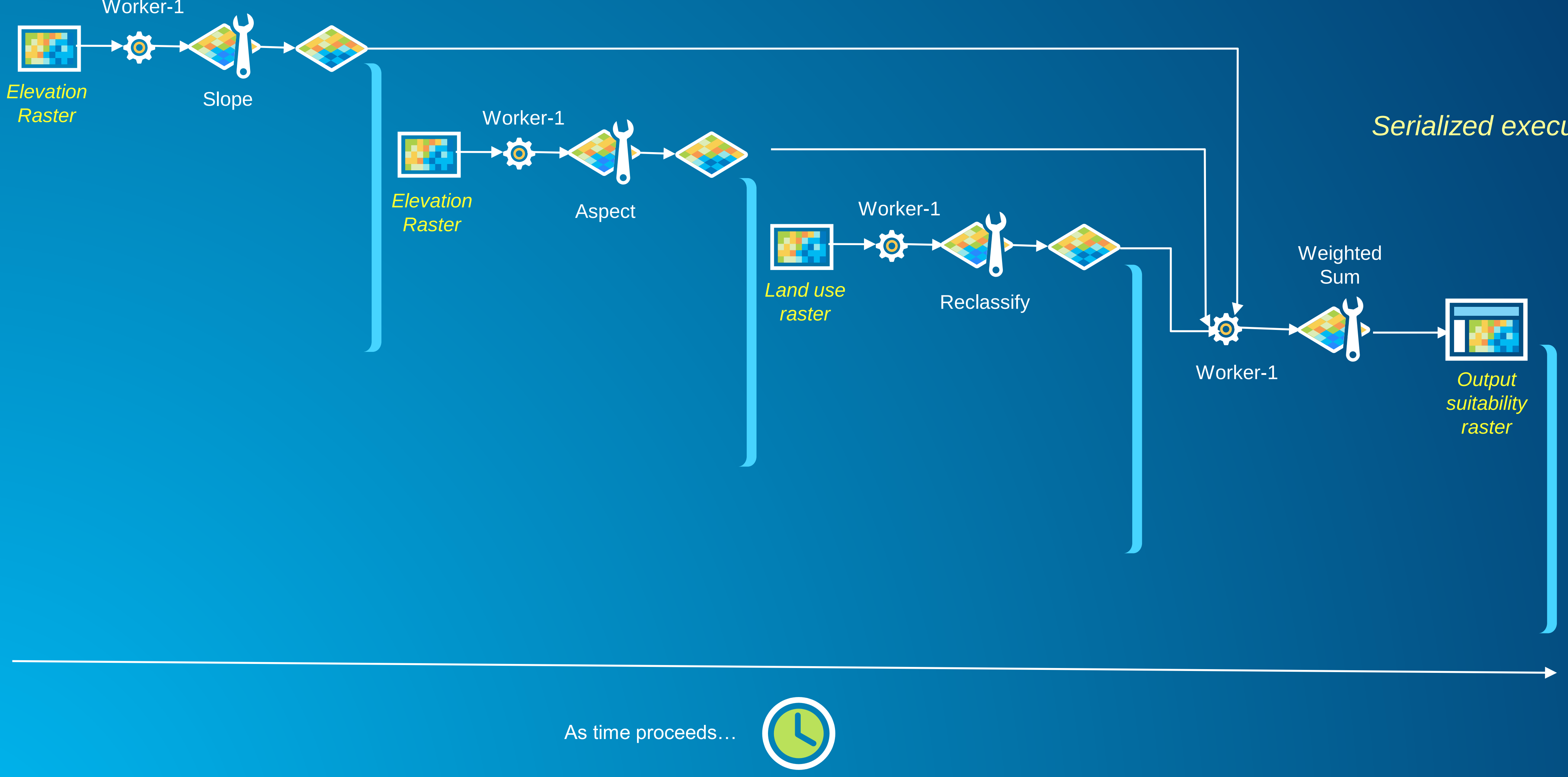
- **“Python is the Language of GIS” – Bill Moreland, Esri Dev Summit 2014**
- **GIS is inherently more complex than Python on its own**
  - **Combining Programming with Geography**
- **Challenges**
  - **Projected Data**
  - **User Interface Considerations**

Working with Rasters in Multiprocessing

Candidate operations for parallelization:

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

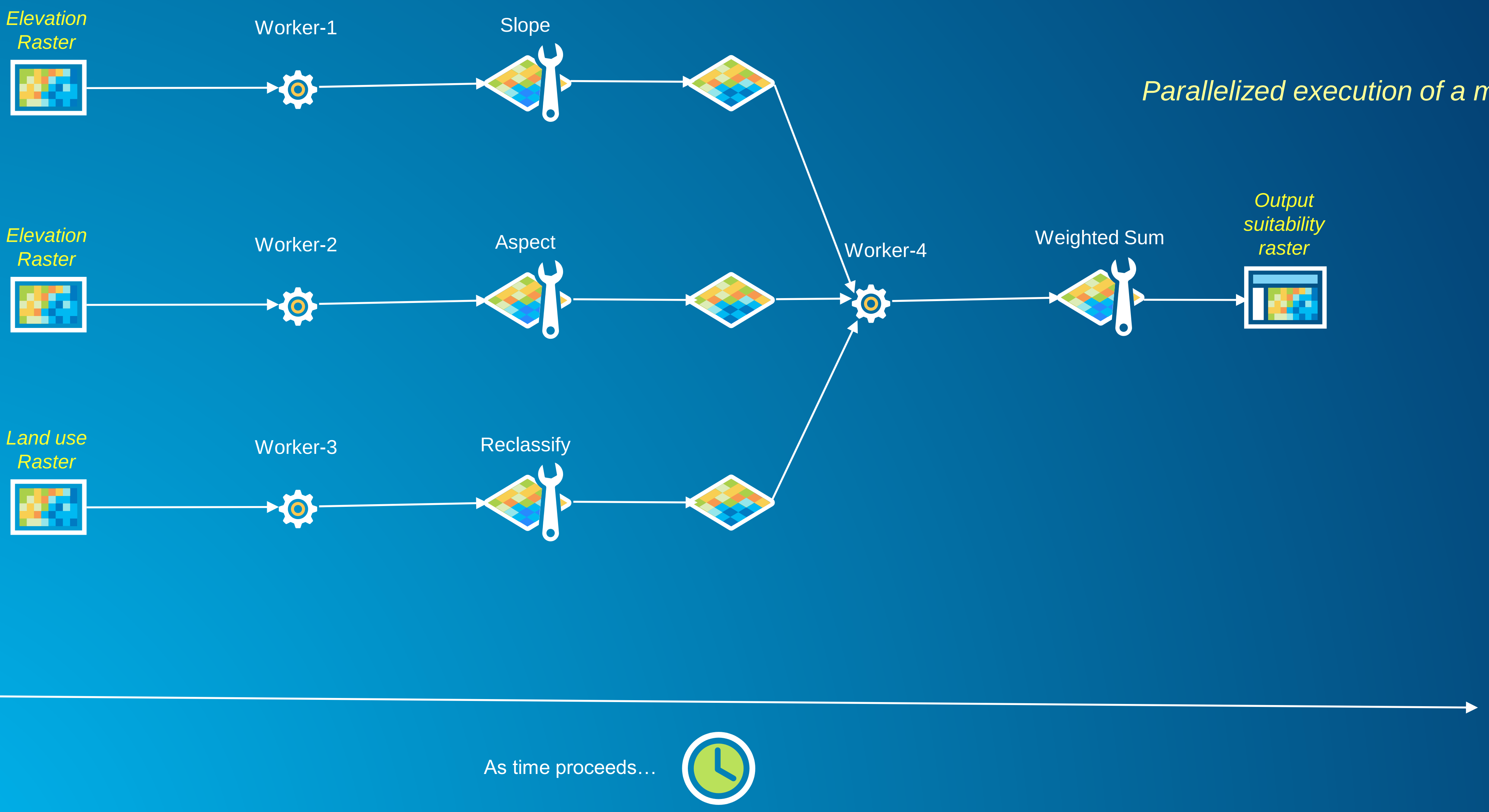
Pleasingly parallel problems



Serialized execution of a model workflow

Why is multiprocessing relevant to Geoprocessing workflows?

Pleasingly parallel problems



Parallelized execution of a model workflow

Why is multiprocessing relevant to Geoprocessing workflows?

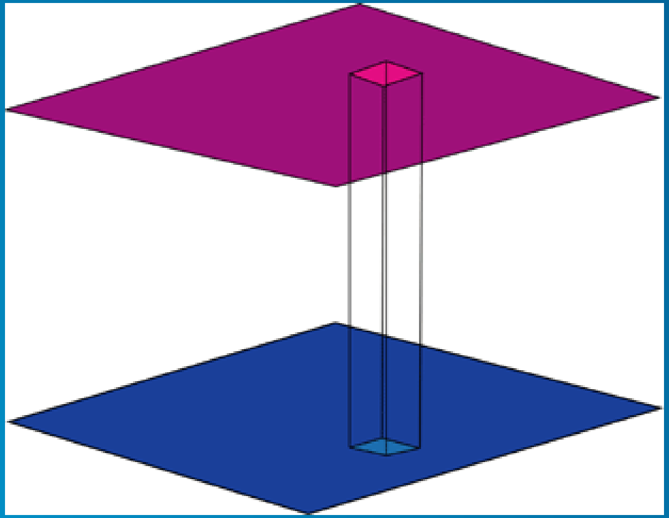
Working with Rasters in Multiprocessing

Candidate operations for parallelization:

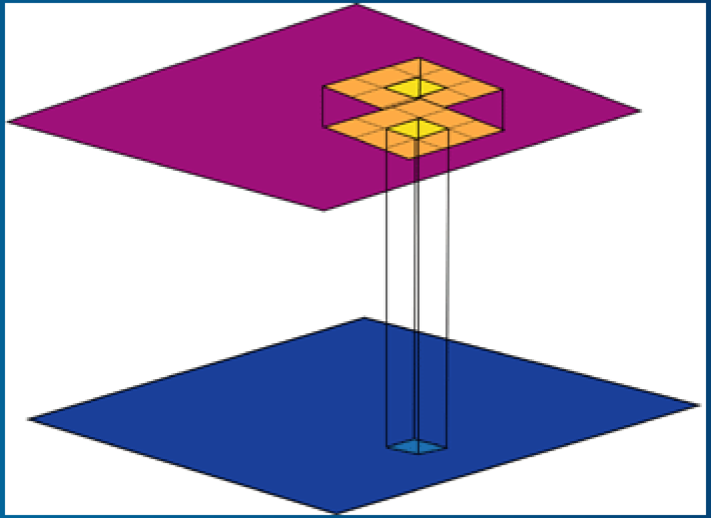
- **Parallelize independent tasks within a workflow**
- **Process a large raster parallelly (limited to local/focal/zonal operations)**

Easy-to-parallelize raster operations

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

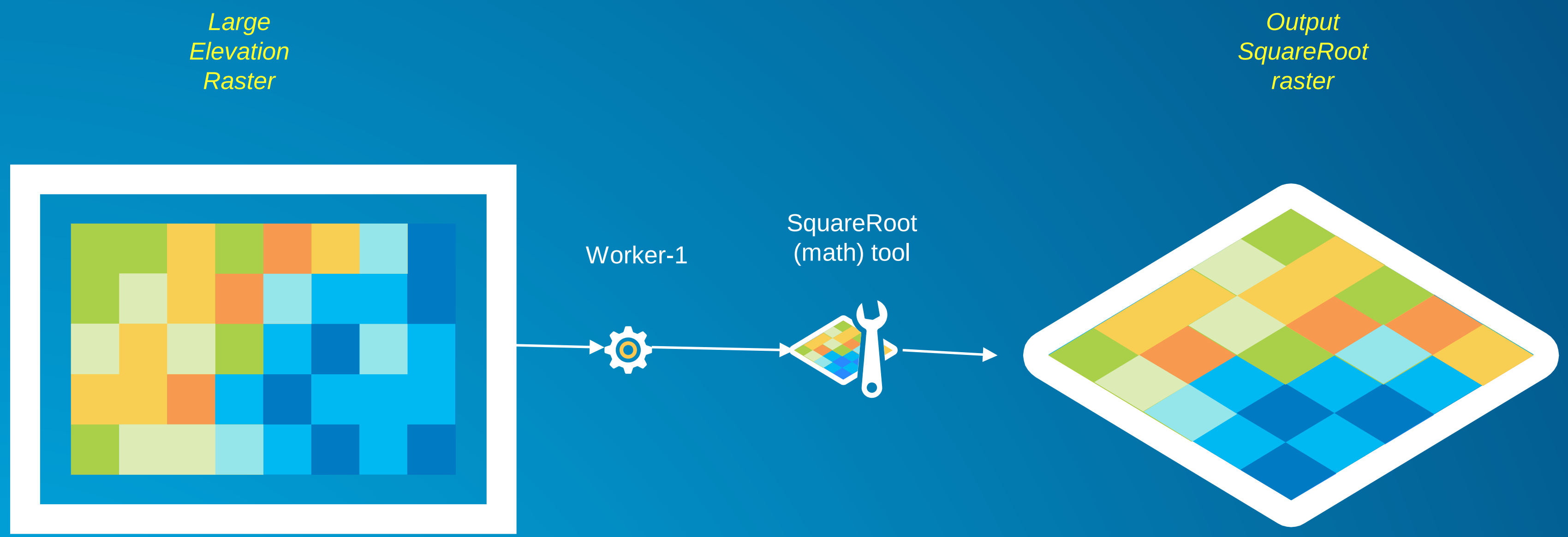


Local raster operation



Focal raster operation

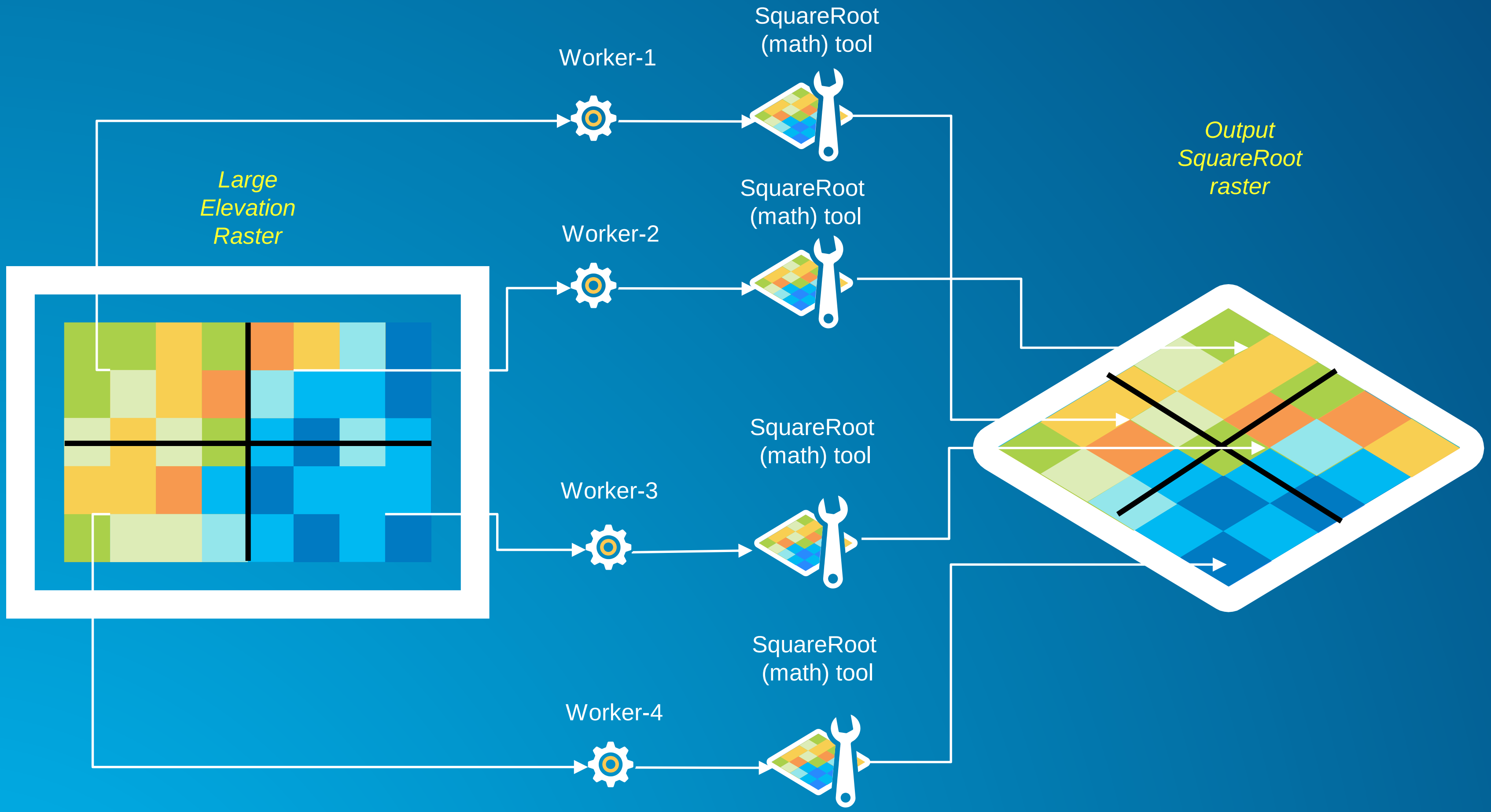
Pleasingly parallel problems



Tool executed serially on a large input dataset

Why is multiprocessing relevant to Geoprocessing workflows?

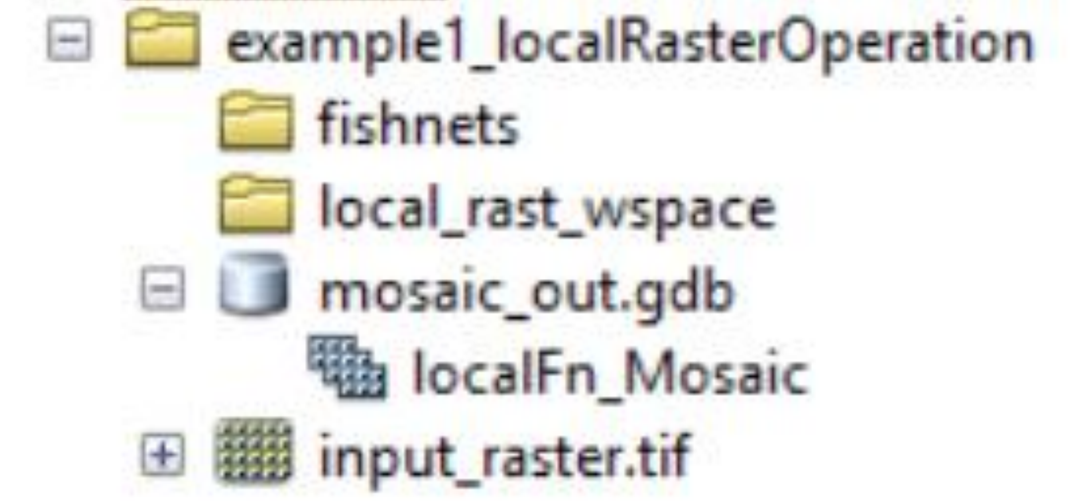
Pleasingly parallel problems



Tool executed parallelly on a large input dataset

Why is multiprocessing relevant to Geoprocessing workflows?

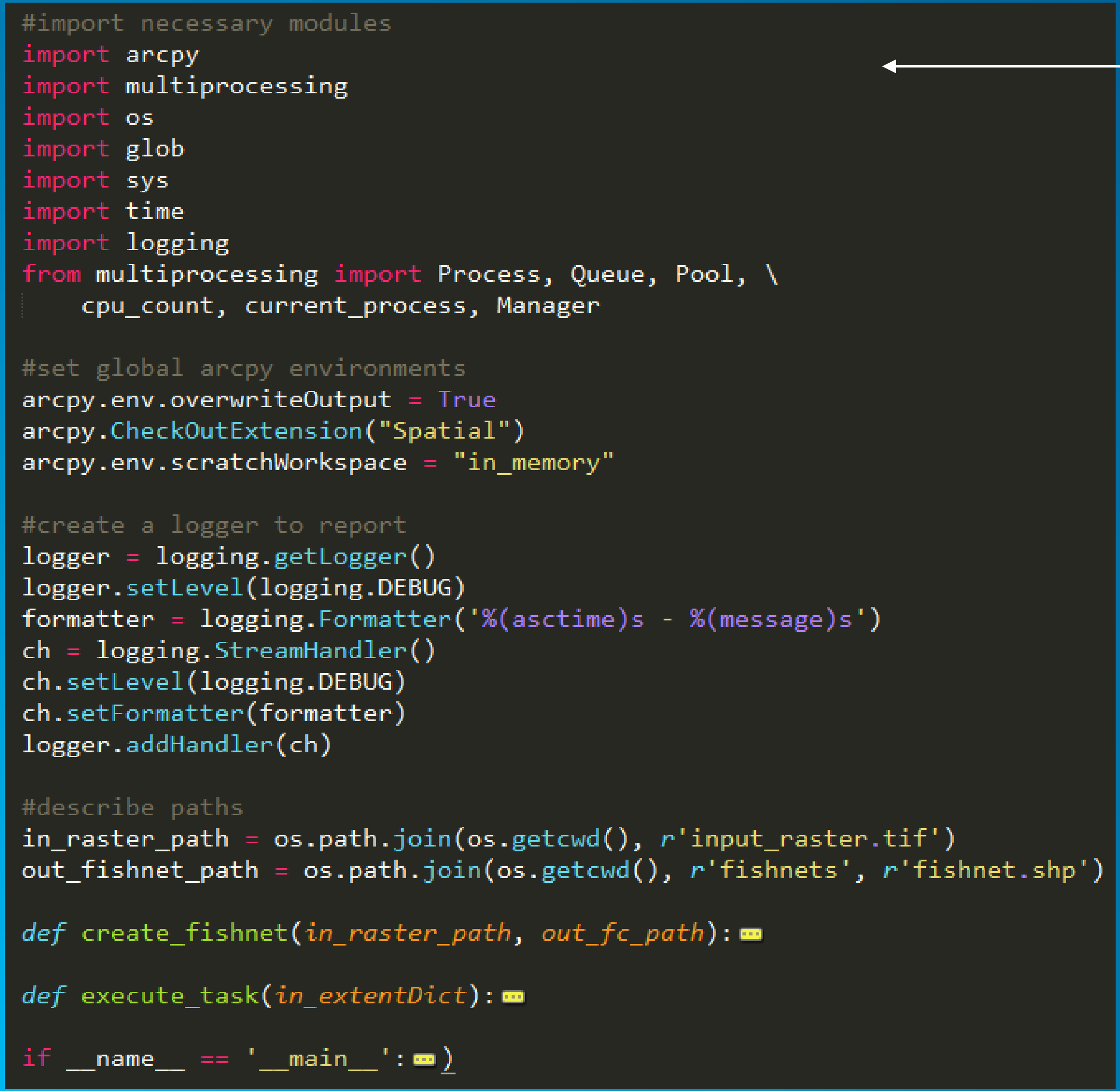
Demo of multiprocessing with Rasters


example1_localRasterOperation
fishnets
local_rast_wspace
mosaic_out.gdb
localFn_Mosaic
input_raster.tif


parallel_local PY File

Run a local raster operation parallelly on a large raster dataset

Demo of multiprocessing with Rasters

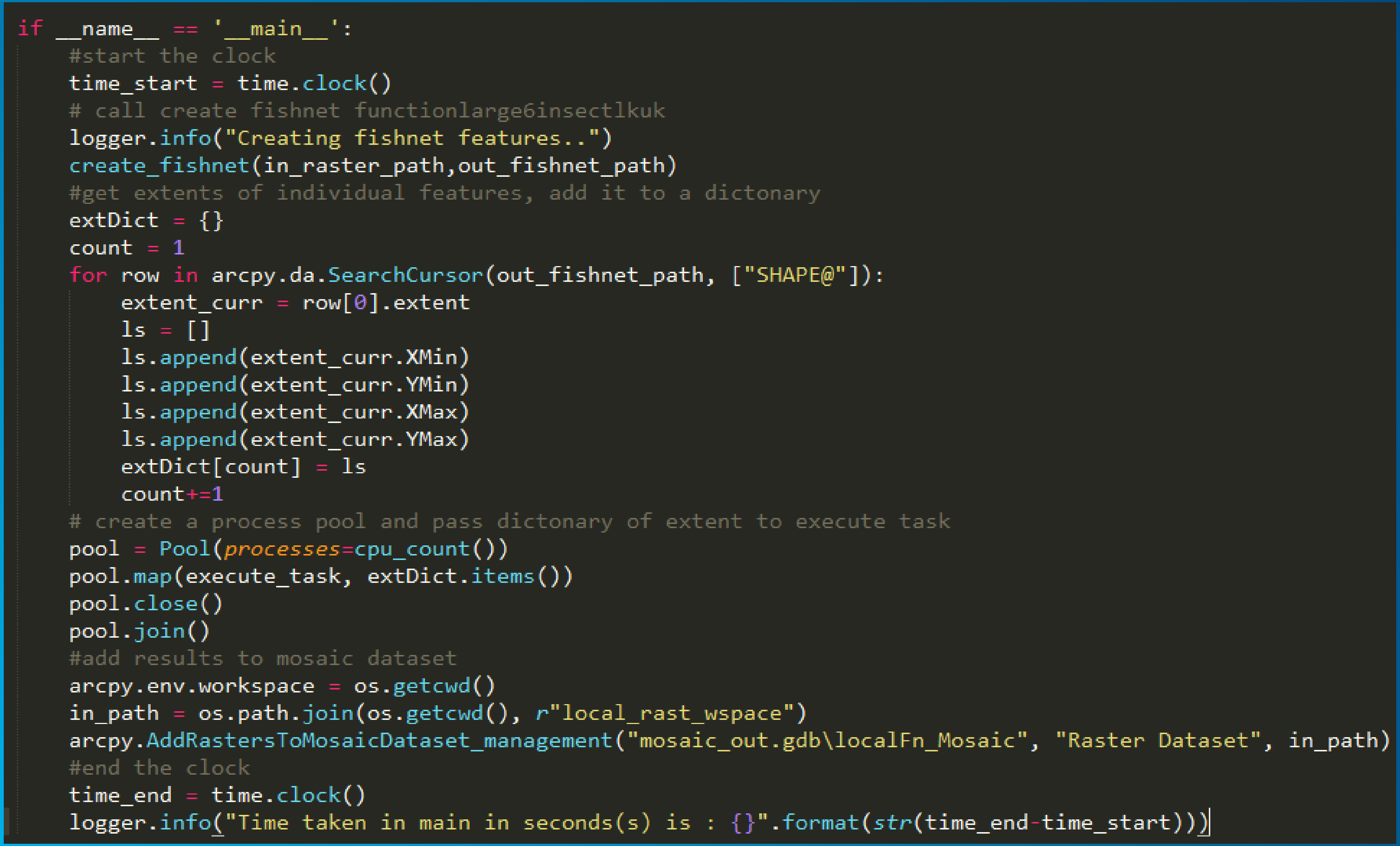


```
#import necessary modules
import arcpy
import multiprocessing
import os
import glob
import sys
import time
import logging
from multiprocessing import Process, Queue, Pool, \
    cpu_count, current_process, Manager

#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__':…)
```

parallel_local PY File

Run a local raster operation parallelly on a large raster dataset

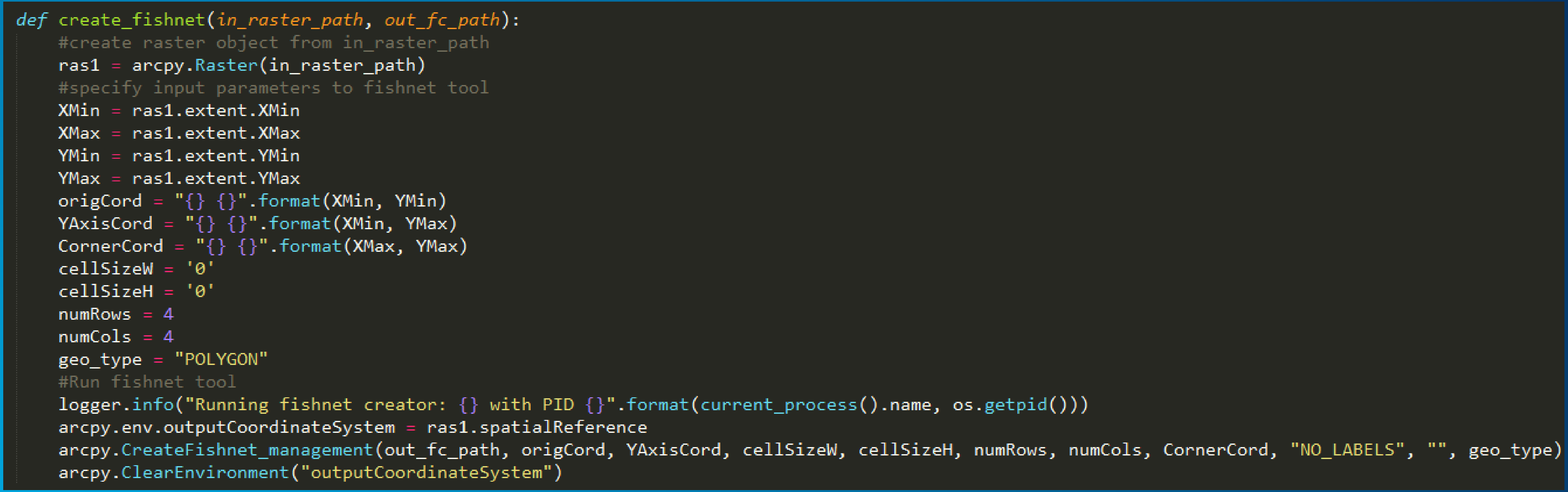
Demo of multiprocessing with Rasters



```
if __name__ == '__main__':
    #start the clock
    time_start = time.clock()
    # call create fishnet functionlarge6insectlkuk
    logger.info("Creating fishnet features..")
    create_fishnet(in_raster_path,out_fishnet_path)
    #get extents of individual features, add it to a dictonary
    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 dictonary 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_wspace")
    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 parallelly on a large raster dataset

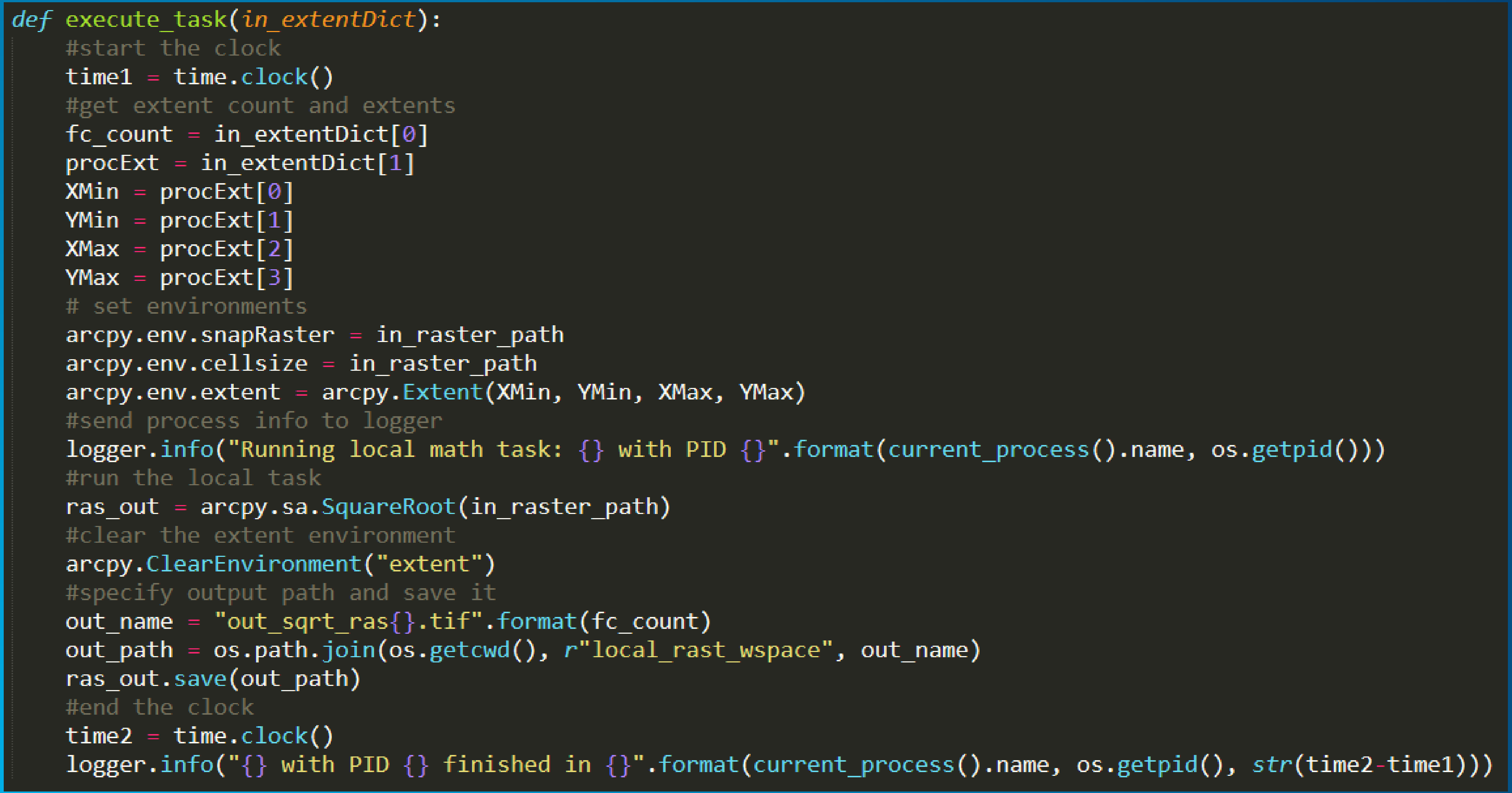
Demo of multiprocessing with Rasters



```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.ClearEnvironment("outputCoordinateSystem")
```

Run a local raster operation parallelly on a large raster dataset

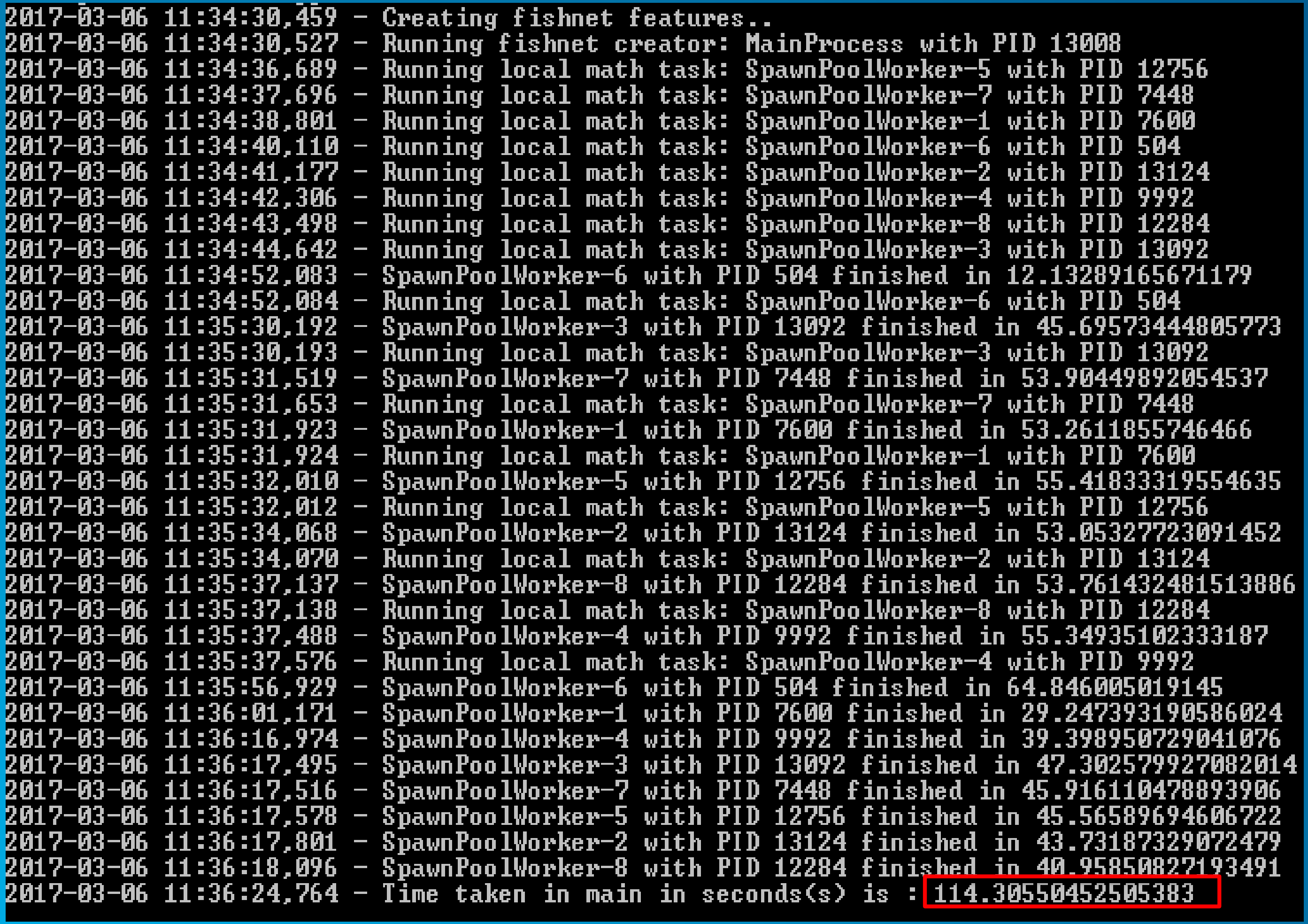
Demo of multiprocessing with Rasters



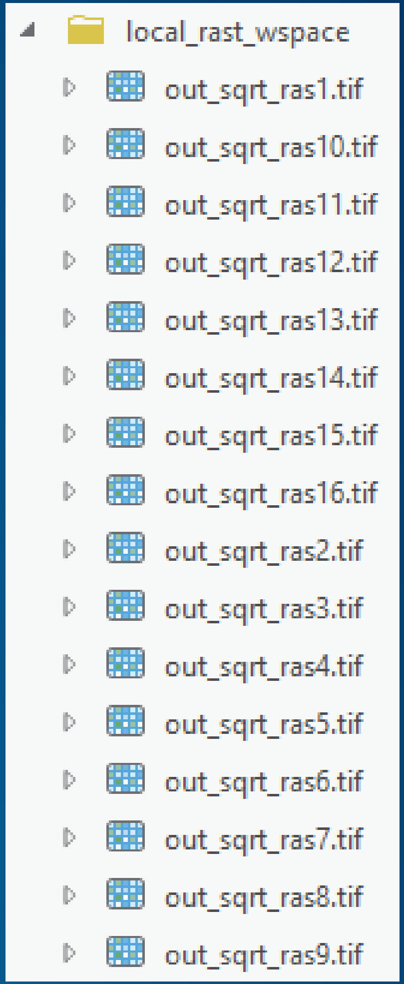
```
def execute_task(in_extentDict):
    #start the clock
    time1 = time.clock()
    #get extent count and extents
    fc_count = in_extentDict[0]
    procExt = in_extentDict[1]
    XMin = procExt[0]
    YMin = procExt[1]
    XMax = procExt[2]
    YMax = procExt[3]
    # set environments
    arcpy.env.snapRaster = 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 PID {}".format(current_process().name, os.getpid()))
    #run the local task
    ras_out = arcpy.sa.SquareRoot(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



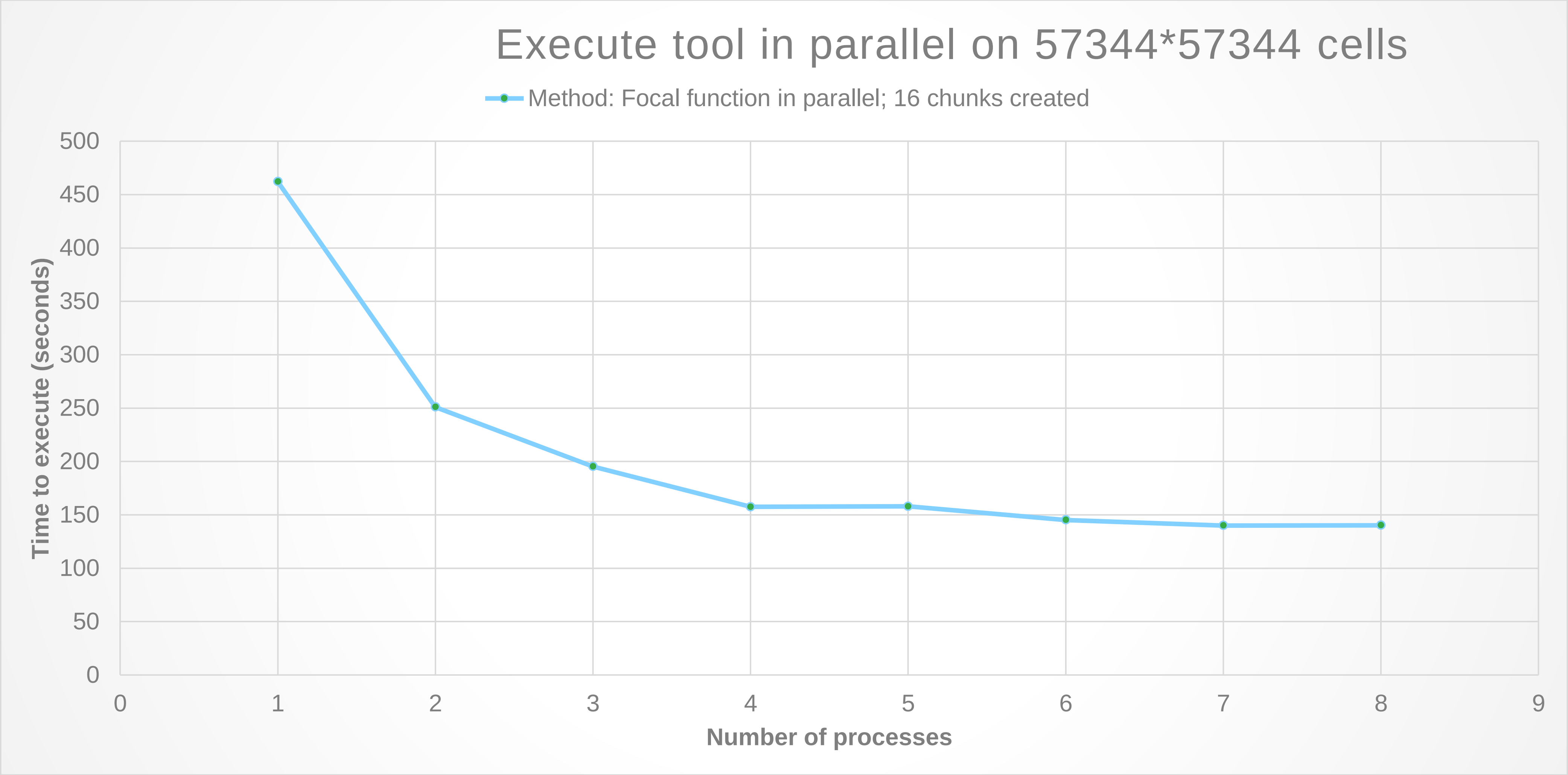
```
2017-03-06 11:34:30,459 - Creating fishnet features..
2017-03-06 11:34:30,527 - Running fishnet creator: MainProcess with PID 13008
2017-03-06 11:34:36,689 - Running local math task: SpawnPoolWorker-5 with PID 12756
2017-03-06 11:34:37,696 - Running local math task: SpawnPoolWorker-7 with PID 7448
2017-03-06 11:34:38,801 - Running local math task: SpawnPoolWorker-1 with PID 7600
2017-03-06 11:34:40,110 - Running local math task: SpawnPoolWorker-6 with PID 504
2017-03-06 11:34:41,177 - Running local math task: SpawnPoolWorker-2 with PID 13124
2017-03-06 11:34:42,306 - Running local math task: SpawnPoolWorker-4 with PID 9992
2017-03-06 11:34:43,498 - Running local math task: SpawnPoolWorker-8 with PID 12284
2017-03-06 11:34:44,642 - Running local math task: SpawnPoolWorker-3 with PID 13092
2017-03-06 11:34:52,083 - SpawnPoolWorker-6 with PID 504 finished in 12.13289165671179
2017-03-06 11:34:52,084 - Running local math task: SpawnPoolWorker-6 with PID 504
2017-03-06 11:35:30,192 - SpawnPoolWorker-3 with PID 13092 finished in 45.69573444805773
2017-03-06 11:35:30,193 - Running local math task: SpawnPoolWorker-3 with PID 13092
2017-03-06 11:35:31,519 - SpawnPoolWorker-7 with PID 7448 finished in 53.90449892054537
2017-03-06 11:35:31,653 - Running local math task: SpawnPoolWorker-7 with PID 7448
2017-03-06 11:35:31,923 - SpawnPoolWorker-1 with PID 7600 finished in 53.2611855746466
2017-03-06 11:35:31,924 - Running local math task: SpawnPoolWorker-1 with PID 7600
2017-03-06 11:35:32,010 - SpawnPoolWorker-5 with PID 12756 finished in 55.41833319554635
2017-03-06 11:35:32,012 - Running local math task: SpawnPoolWorker-5 with PID 12756
2017-03-06 11:35:34,068 - SpawnPoolWorker-2 with PID 13124 finished in 53.05327723091452
2017-03-06 11:35:34,070 - Running local math task: SpawnPoolWorker-2 with PID 13124
2017-03-06 11:35:37,137 - SpawnPoolWorker-8 with PID 12284 finished in 53.761432481513886
2017-03-06 11:35:37,138 - Running local math task: SpawnPoolWorker-8 with PID 12284
2017-03-06 11:35:37,488 - SpawnPoolWorker-4 with PID 9992 finished in 55.34935102333187
2017-03-06 11:35:37,576 - Running local math task: SpawnPoolWorker-4 with PID 9992
2017-03-06 11:35:56,929 - SpawnPoolWorker-6 with PID 504 finished in 64.846005019145
2017-03-06 11:36:01,171 - SpawnPoolWorker-1 with PID 7600 finished in 29.247393190586024
2017-03-06 11:36:16,974 - SpawnPoolWorker-4 with PID 9992 finished in 39.398950729041076
2017-03-06 11:36:17,495 - SpawnPoolWorker-3 with PID 13092 finished in 47.302579927082014
2017-03-06 11:36:17,516 - SpawnPoolWorker-7 with PID 7448 finished in 45.916110478893906
2017-03-06 11:36:17,578 - SpawnPoolWorker-5 with PID 12756 finished in 45.56589694606722
2017-03-06 11:36:17,801 - SpawnPoolWorker-2 with PID 13124 finished in 43.73187329072479
2017-03-06 11:36:18,096 - SpawnPoolWorker-8 with PID 12284 finished in 40.95850827193491
2017-03-06 11:36:24,764 - Time taken in main in seconds(s) is : 114.30550452505383
```



local_rast_wspace
out_sqrt_ras1.tif
out_sqrt_ras10.tif
out_sqrt_ras11.tif
out_sqrt_ras12.tif
out_sqrt_ras13.tif
out_sqrt_ras14.tif
out_sqrt_ras15.tif
out_sqrt_ras16.tif
out_sqrt_ras2.tif
out_sqrt_ras3.tif
out_sqrt_ras4.tif
out_sqrt_ras5.tif
out_sqrt_ras6.tif
out_sqrt_ras7.tif
out_sqrt_ras8.tif
out_sqrt_ras9.tif

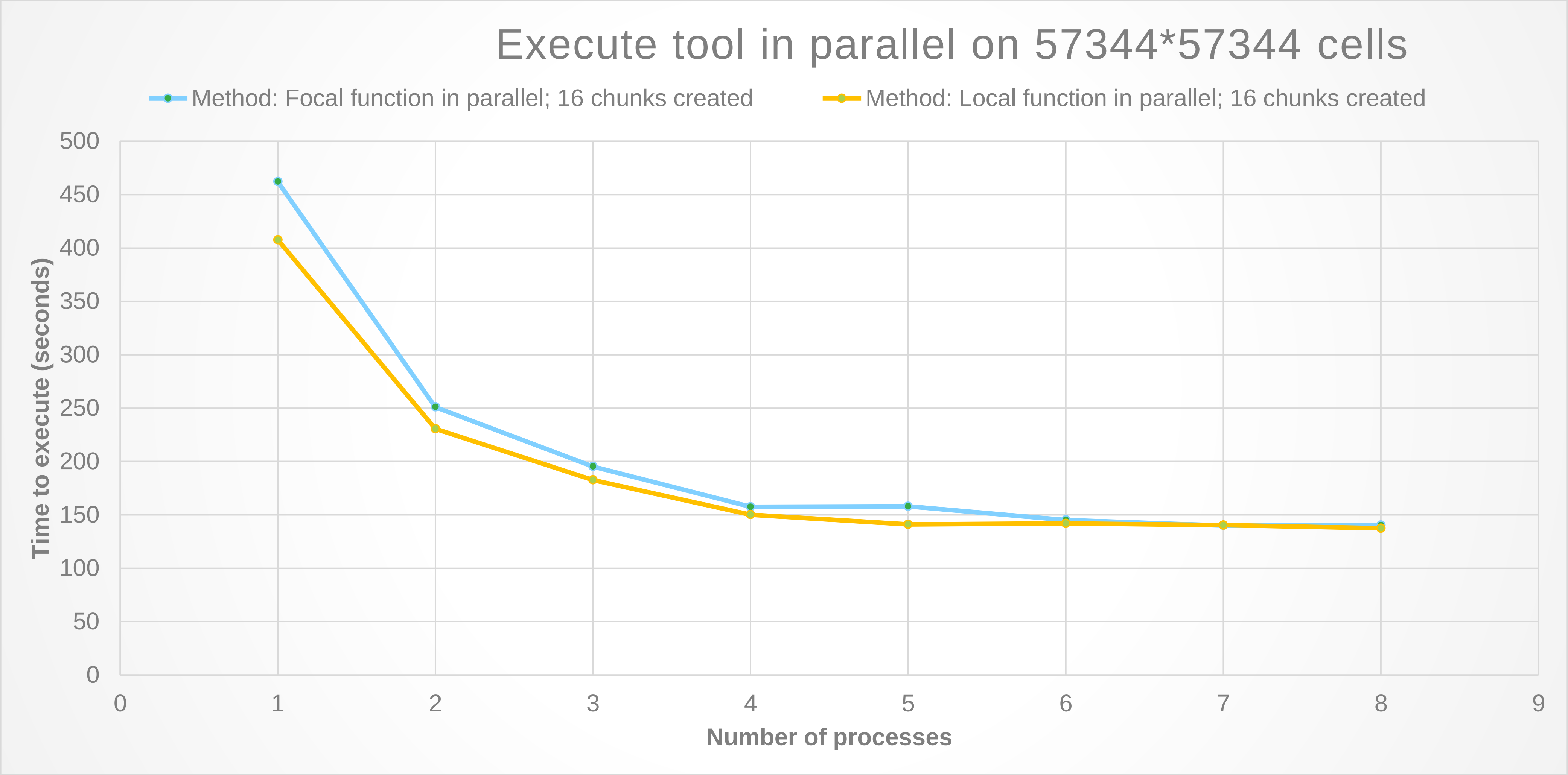
Run a local raster operation parallelly on a large raster dataset

Raster analysis performance improvements with multiprocessing



Run a local raster operation parallelly on a large raster dataset

Raster analysis performance improvements with multiprocessing

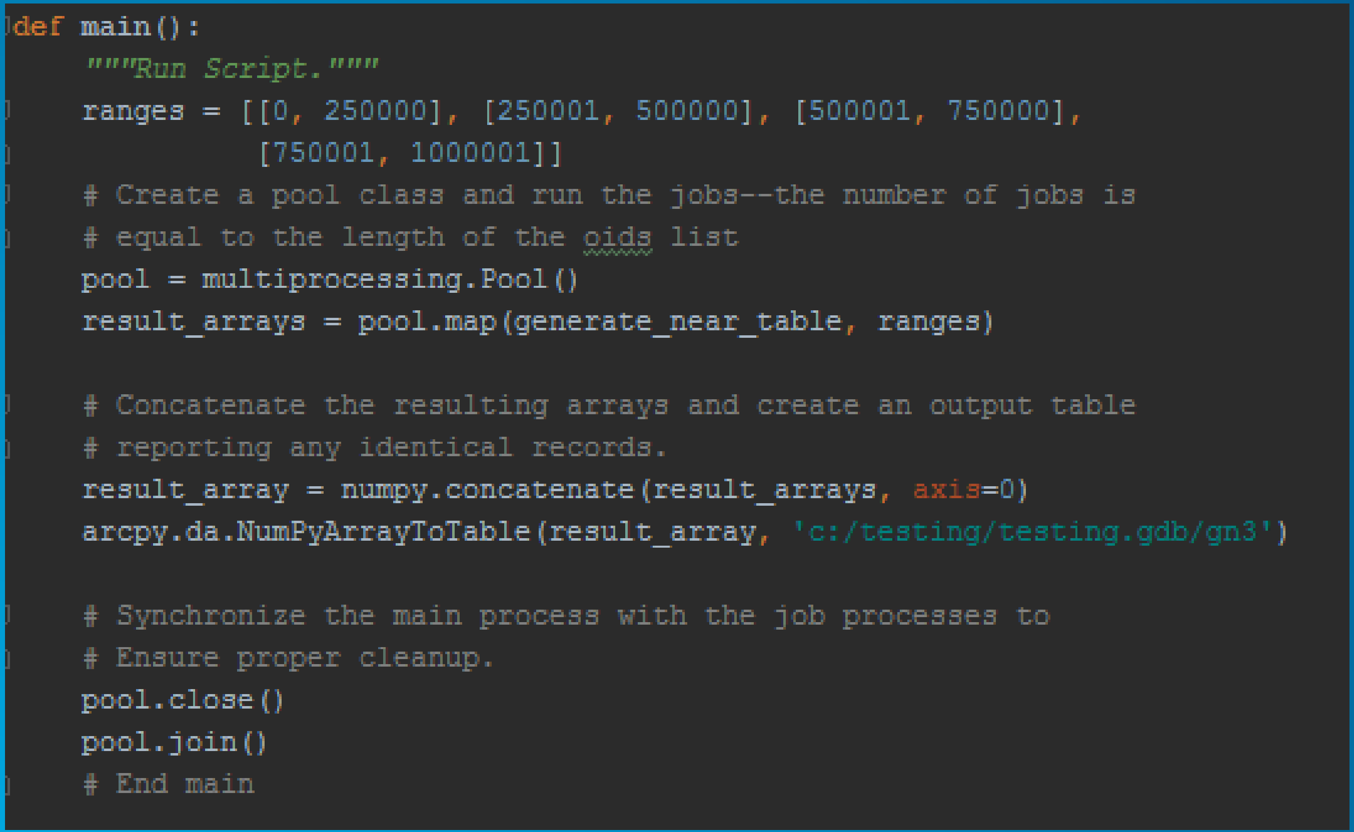


Run a local raster operation parallelly on a large raster dataset

Working with Vectors in Multiprocessing

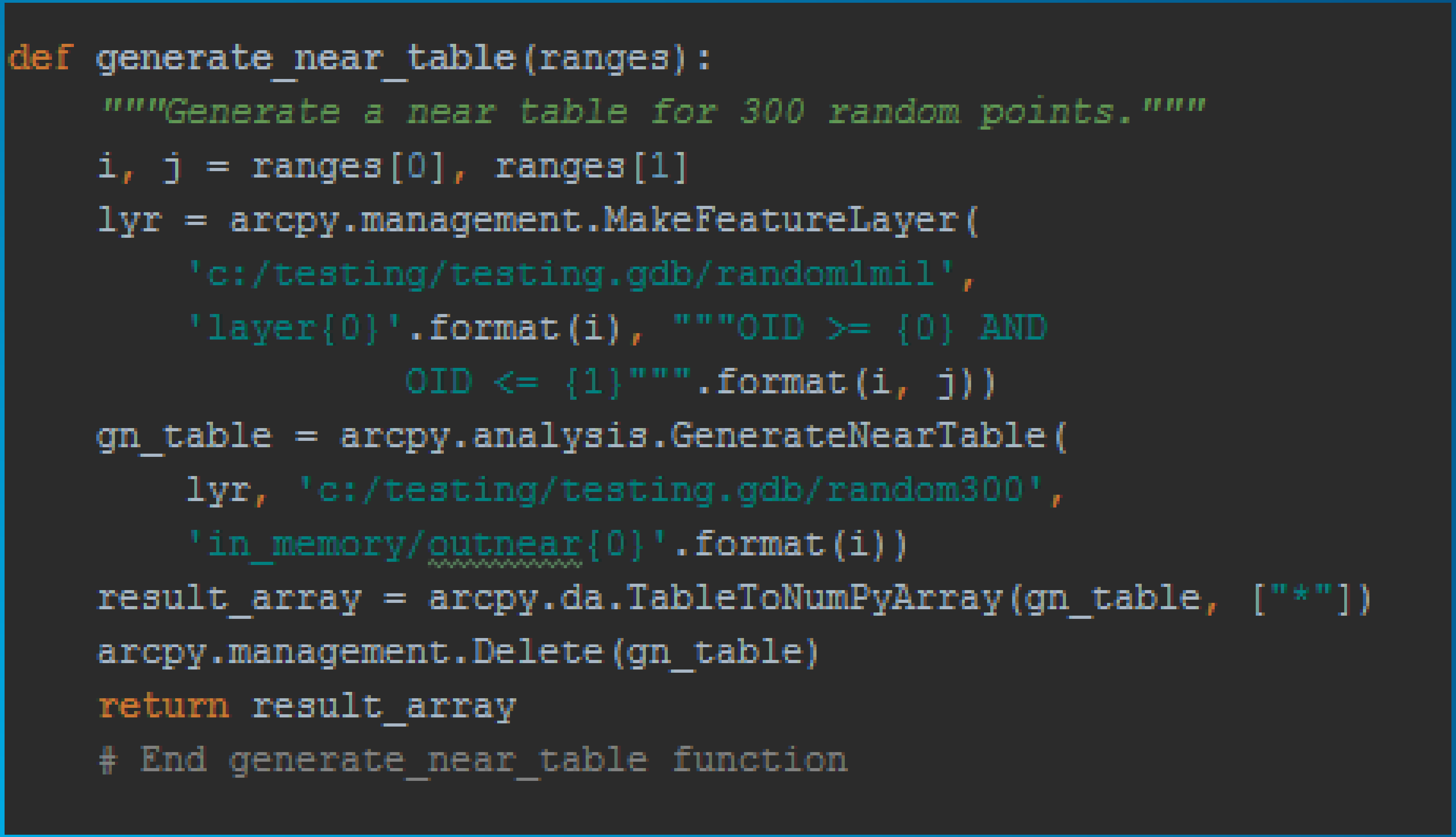
- Vectors are discrete units of data
  - Concept aligns easily with multiprocessing
  - Each unit of data can be mapped to an independent process
  - To aggregate return results:
    - Independent processes return results to a collection
    - Minimal amount of functionality on aggregation step
    - 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



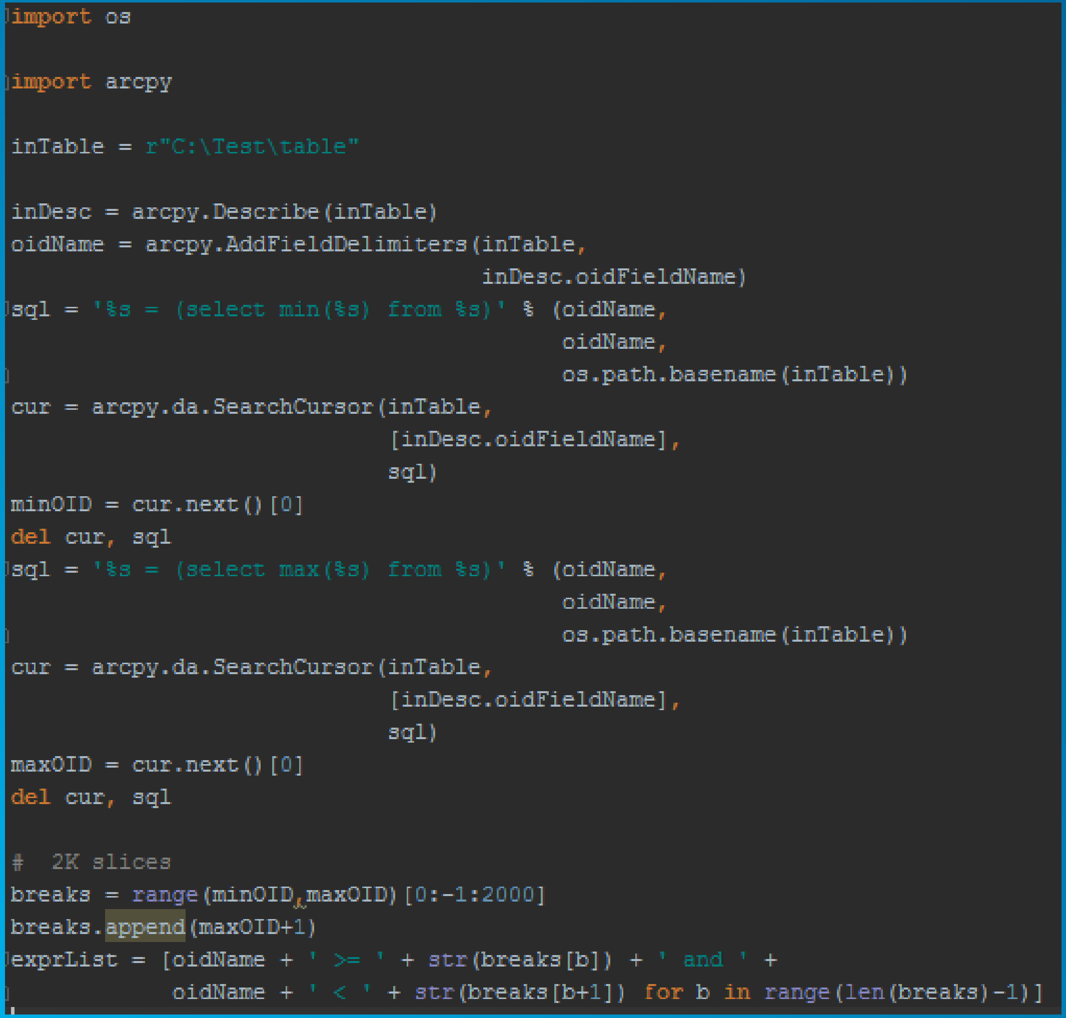
```
def main():
    """Run Script."""
    ranges = [[0, 250000], [250001, 500000], [500001, 750000],
              [750001, 1000001]]
    # Create a pool class and run the jobs--the number of jobs is
    # equal to the length of the oids list
    pool = multiprocessing.Pool()
    result_arrays = pool.map(generate_near_table,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

Example of multiprocessing with Vectors



```
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{0}'.format(i), """OID >= {0} AND
                  OID <= {1}""".format(i, j))
    gn_table = arcpy.analysis.GenerateNearTable(
        lyr, 'c:/testing/testing.gdb/random300',
        'in_memory/outnear{0}'.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



```
import os

import arcpy

inTable = r"C:\Test\table"

inDesc = arcpy.Describe(inTable)
oidName = arcpy.AddFieldDelimiters(inTable,
                                   inDesc.oidFieldName)
sql = '%s = (select min(%s) from %s)' % (oidName,
                                         oidName,
                                         os.path.basename(inTable))
cur = arcpy.da.SearchCursor(inTable,
                            [inDesc.oidFieldName],
                            sql)
minOID = cur.next()[0]
del cur, sql
sql = '%s = (select max(%s) from %s)' % (oidName,
                                         oidName,
                                         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 = [oidName + ' >= ' + str(breaks[b]) + ' and ' +
            oidName + ' < ' + str(breaks[b+1]) for b in range(len(breaks)-1)]
```

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**
  - http://dask.pydata.org/en/latest/

- **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**
- **Esri blog post on parallel geoprocessing - https://blogs.esri.com/esri/arcgis/2012/09/26/distributed-processing-with-arcgis-part-1/**
- **Python 3.5 multiprocessing API- https://docs.python.org/3.5/library/multiprocessing.html**

esri

THE
SCIENCE
OF
WHERE