ArcPy: Solving Large Transportation Analysis Problems

Deelesh Mandloi & Melinda Morang

2020 ESRI DEVELOPER SUMMIT | Palm Springs, CA

Agenda

- Intro/background
- Techniques for solving large problems
- Python script walk-through (solving locally)
- Working with services
- Python script walk-through (solving with a service)

Code and slides:

http://esriurl.com/ds20pyslnp

Introduction

ArcGIS Network Analyst Extension

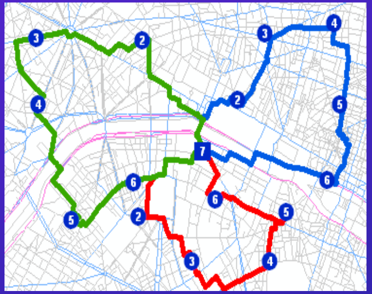
for transportation analysis

Coverage

Service Area

Optimization

Location-Allocation

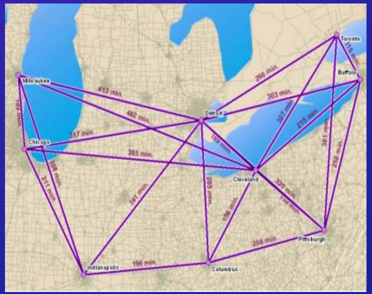


Vehicle Routing Problem

Point-to-point routing

Route

Closest Facility



Origin-Destination Cost Matrix

OD Cost Matrix

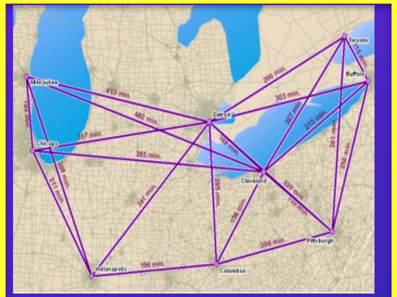
Calculates the **travel time** or **distance** between a **set of origins** and a **set of destinations**

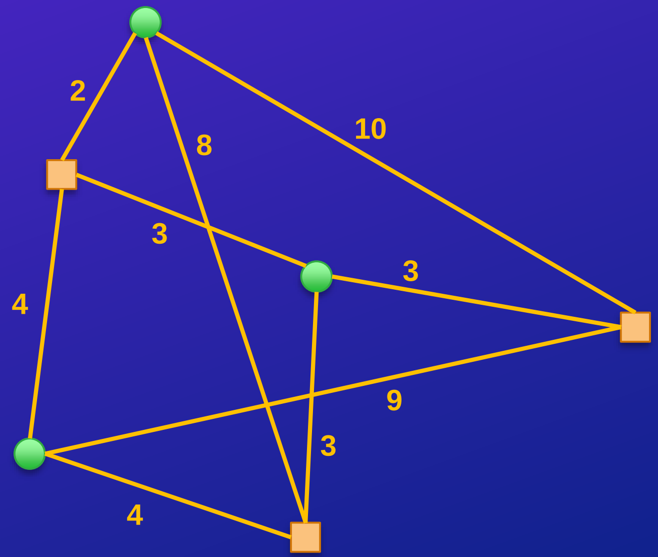
- Can use a time/distance limit
- Can limit the number of destinations to find

Point-to-point routing

Route

Closest Facility


Origin-Destination Cost Matrix



What is a large problem?

- Can’t be solved in one calculation
  - Unreasonable calculation time
  - Memory limits
  - Service limits
- Large number of inputs
- Large number of outputs
  - number of origins x number of destinations
  - 1000 x 1000 = 1,000,000

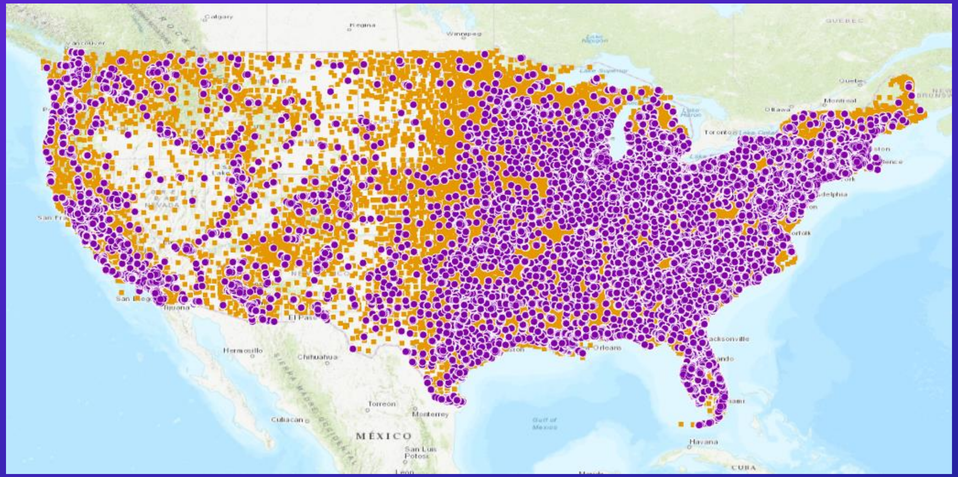
Examples

- Calculate drive time for all patients to all medical clinics within 100 miles
- Calculate the network distance from every parcel to every other parcel

Today’s goal

Solve large OD Cost Matrix

- Any number of origins and destinations
- Using local data or a service
- With or without a time/distance limit
- Output a single feature class



What is a local solve and a service solve?

- Local solve
  - Solving a network analysis on your computer using a network dataset stored on your disk
  - Solve uses computing resources of your computer
- Service solve
  - Solving a network analysis using a GIS web service
  - Solve uses computing resources of GIS Server

Which product should I use?

ArcGIS Pro

- 64-bit
- Seamless integration with services
- arcpy.nax module

ArcMap

- 32-bit…runs out of memory easily
- For 64-bit capabilities, must use ArcGIS Server or the 64-bit Background Geoprocessing Extension
- Less simple to use services
- No arcpy.nax module

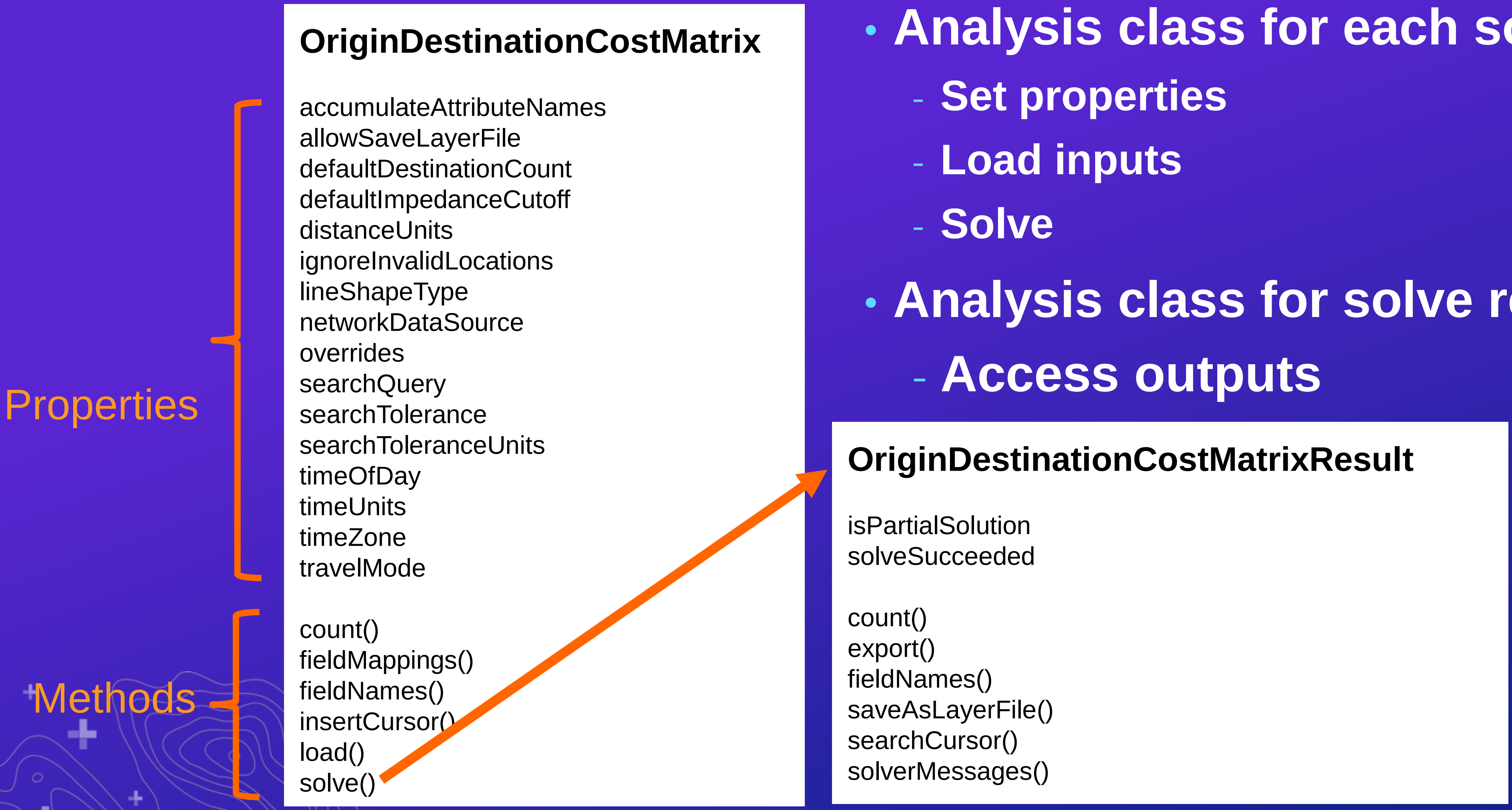
Network Analysis Workflow with arcpy.nax

1. **Initialize the analysis object (based on a specific network data source)**
2. **Set the properties for the analysis**
3. **Load the inputs**
4. **Solve the analysis**
5. **Work with the results**

Common to all the network analyses

arcpy.nax Analysis (Solver) Classes

- Easy-to-use python objects for network analysis
- Added in ArcGIS Pro 2.4
- Analysis class for each solver
  - Set properties
  - Load inputs
  - Solve
- Analysis class for solve results
  - Access outputs



Properties

Methods

OriginDestinationCostMatrix

accumulateAttributeNames
allowSaveLayerFile
defaultDestinationCount
defaultImpedanceCutoff
distanceUnits
ignoreInvalidLocations
lineShapeType
networkDataSource
overrides
searchQuery
searchTolerance
searchToleranceUnits
timeOfDay
timeUnits
timeZone
travelMode

count()
fieldMappings()
fieldNames()
insertCursor()
load()
solve()

OriginDestinationCostMatrixResult

isPartialSolution
solveSucceeded

count()
export()
fieldNames()
saveAsLayerFile()
searchCursor()
solverMessages()

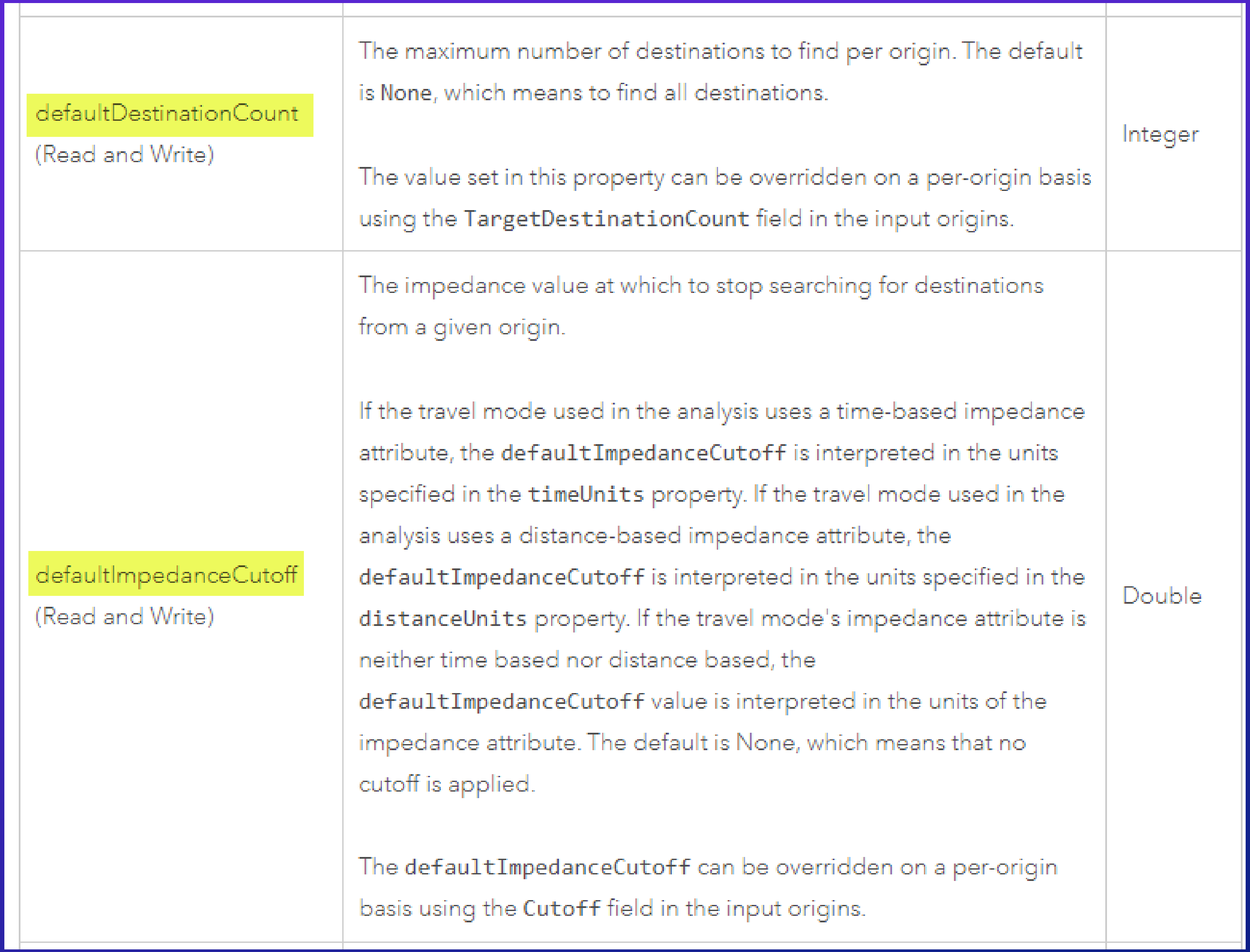
Techniques for solving large problems

How to optimize solving a large problem

- Reduce problem size
- Eliminate irrelevant data
- Chunk data
- Spatially sort data
- Solve in parallel
- Pre-calculate network location fields
- Use network dataset layer

Reduce problem size

- Find only the K nearest
  - defaultDestinationCount property
- Use a time/distance limit
  - Use only destinations within a reasonable straight-line distance of origins
  - defaultImpedanceCutoff property

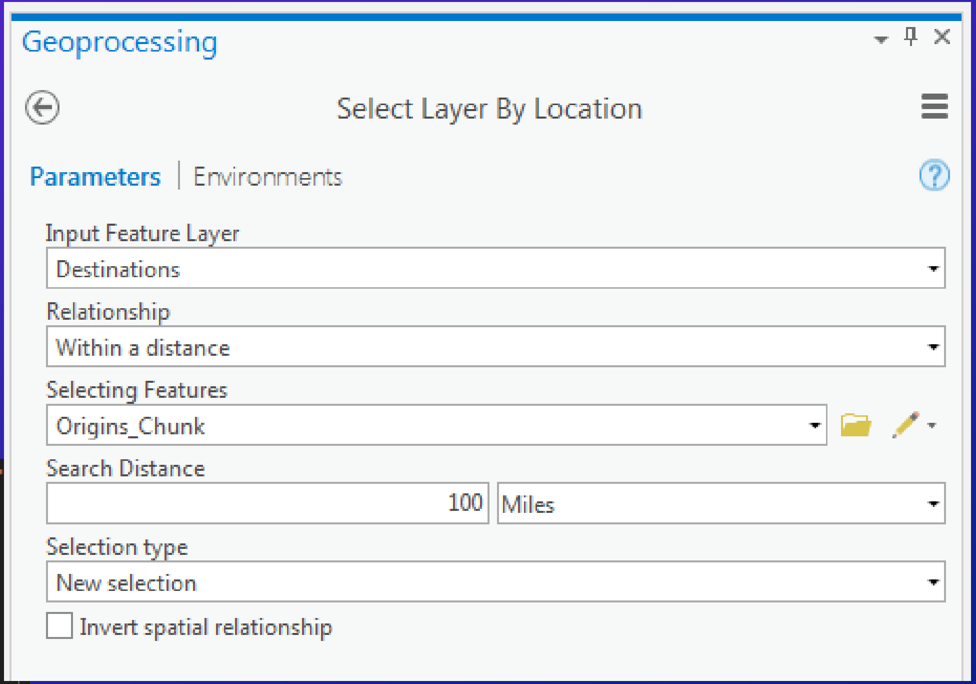


defaultDestinationCount (Read and Write)	The maximum number of destinations to find per origin. The default is None, which means to find all destinations. The value set in this property can be overridden on a per-origin basis using the `TargetDestinationCount` field in the input origins.	Integer
defaultImpedanceCutoff (Read and Write)	The impedance value at which to stop searching for destinations from a given origin. If the travel mode used in the analysis uses a time-based impedance attribute, the `defaultImpedanceCutoff` is interpreted in the units specified in the `timeUnits` property. If the travel mode used in the analysis uses a distance-based impedance attribute, the `defaultImpedanceCutoff` is interpreted in the units specified in the `distanceUnits` property. If the travel mode's impedance attribute is neither time based nor distance based, the `defaultImpedanceCutoff` value is interpreted in the units of the impedance attribute. The default is None, which means that no cutoff is applied. The `defaultImpedanceCutoff` can be overridden on a per-origin basis using the `Cutoff` field in the input origins.	Double

Eliminate irrelevant data

- Remove destinations that aren’t within a reasonable straight-line distance of origins prior to running the OD Cost Matrix analysis
- Applies only if you’re using a time/distance limit
- Use Select Layer By Location
- Watch out for the case where none are selected

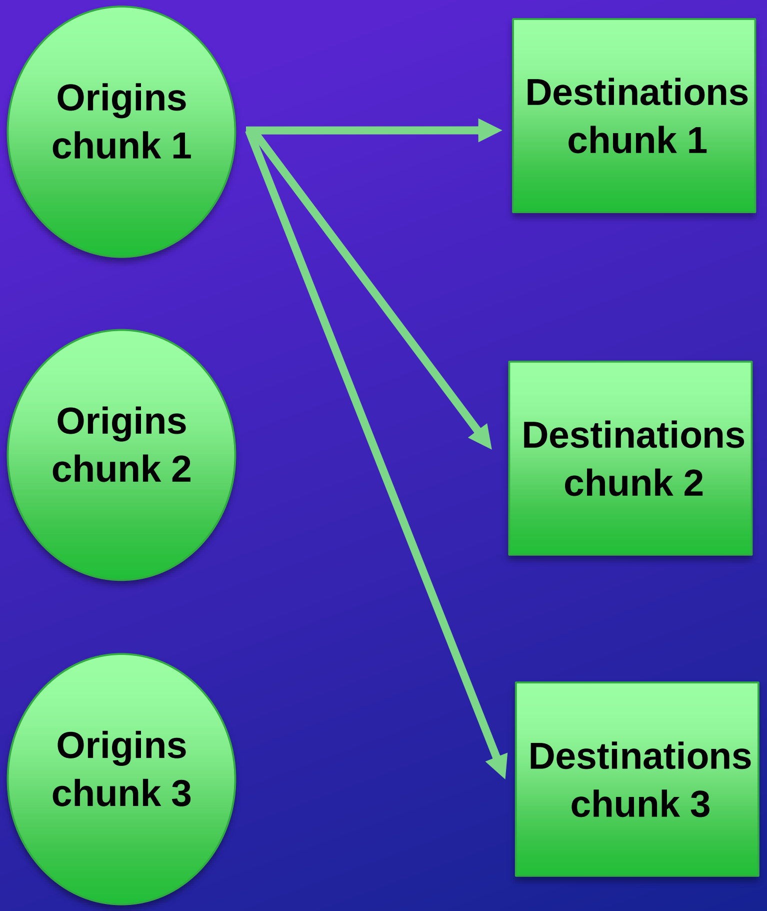
```
result = arcpy.management.SelectLayerByLocation(self.input_destinations_layer, "WITHIN_A_DISTANCE_GEODESIC"
                                                self.input_origins_layer, "{} Miles".format(cutoff),)
# If no destinations are within the cutoff, skip this iteration
if not result.getOutput(0).getSelectionSet():
    msg = "No destinations found within the cutoff"
    self.logger.warning(msg)
    self.job_result["solveMessages"] = msg
    return
```

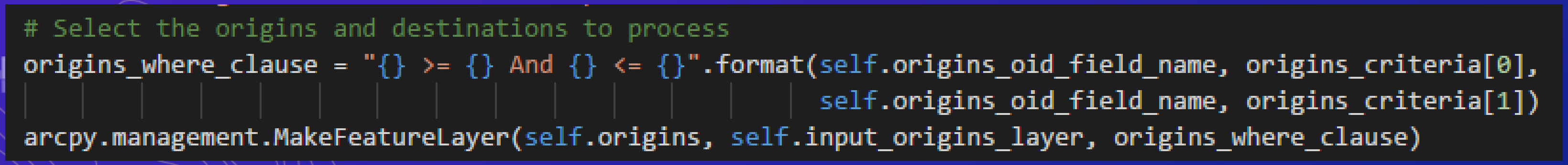


Input Feature Layer: Destinations
Relationship: Within a distance
Selecting Features: Origins_Chunk
Search Distance: 100 Miles
Selection type: New selection
☐ Invert spatial relationship

Chunk data

- Break up origins and destinations into chunks of reasonable size
- Iteratively solve each chunk
- Chunk size depends on service limits or memory limits
- Consider number of origins x number of destinations

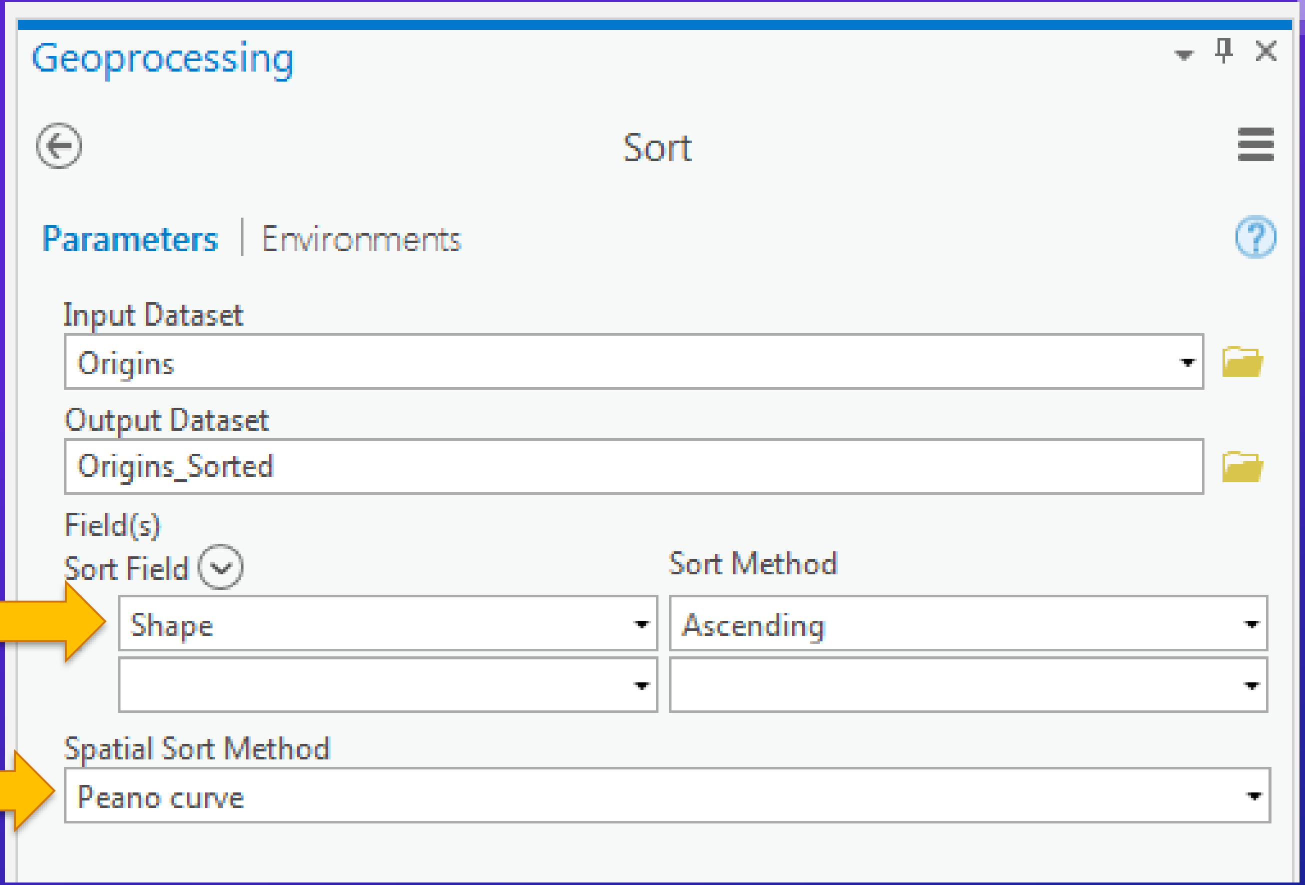




```
# Select the origins and destinations to process
origins_where_clause = "{} >= {} And {} <= {}".format(self.origins_oid_field_name, origins_criteria[0],
                                                      self.origins_oid_field_name, origins_criteria[1])
arcpy.management.MakeFeatureLayer(self.origins, self.input_origins_layer, origins_where_clause)
```

Spatially sort data

- Sort geoprocessing tool
- Sort by Shape field using Peano curve
- Requires Advanced license


Geoprocessing

Sort

Parameters | Environments

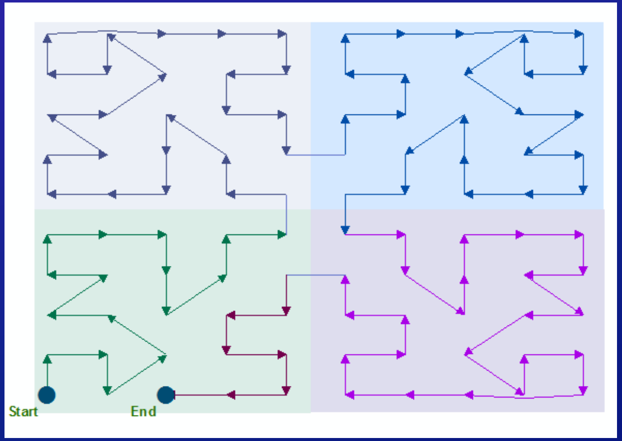
Input Dataset: Origins

Output Dataset: Origins_Sorted

Field(s)

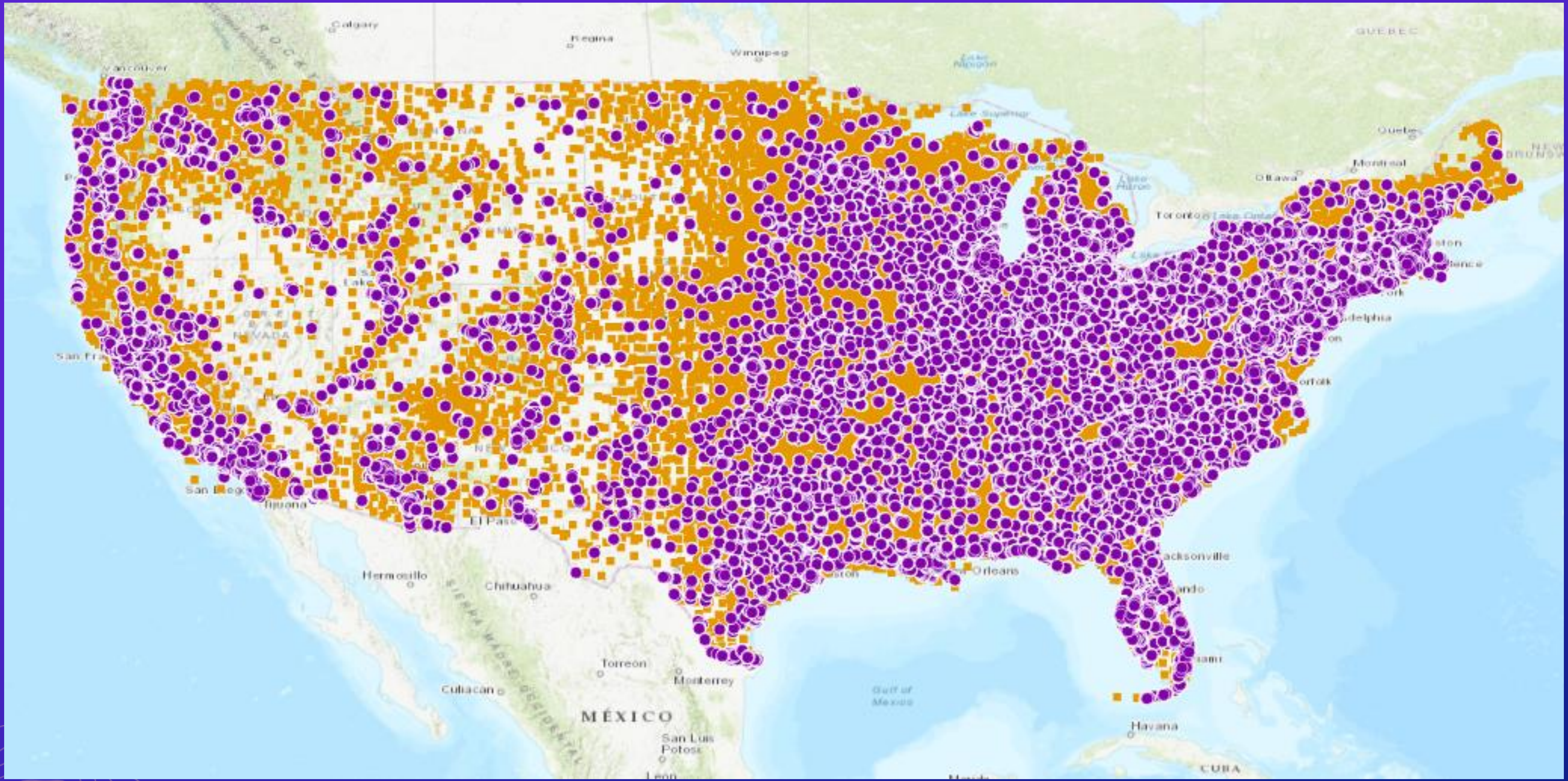
Sort Field	Sort Method
Shape	Ascending

Spatial Sort Method: Peano curve



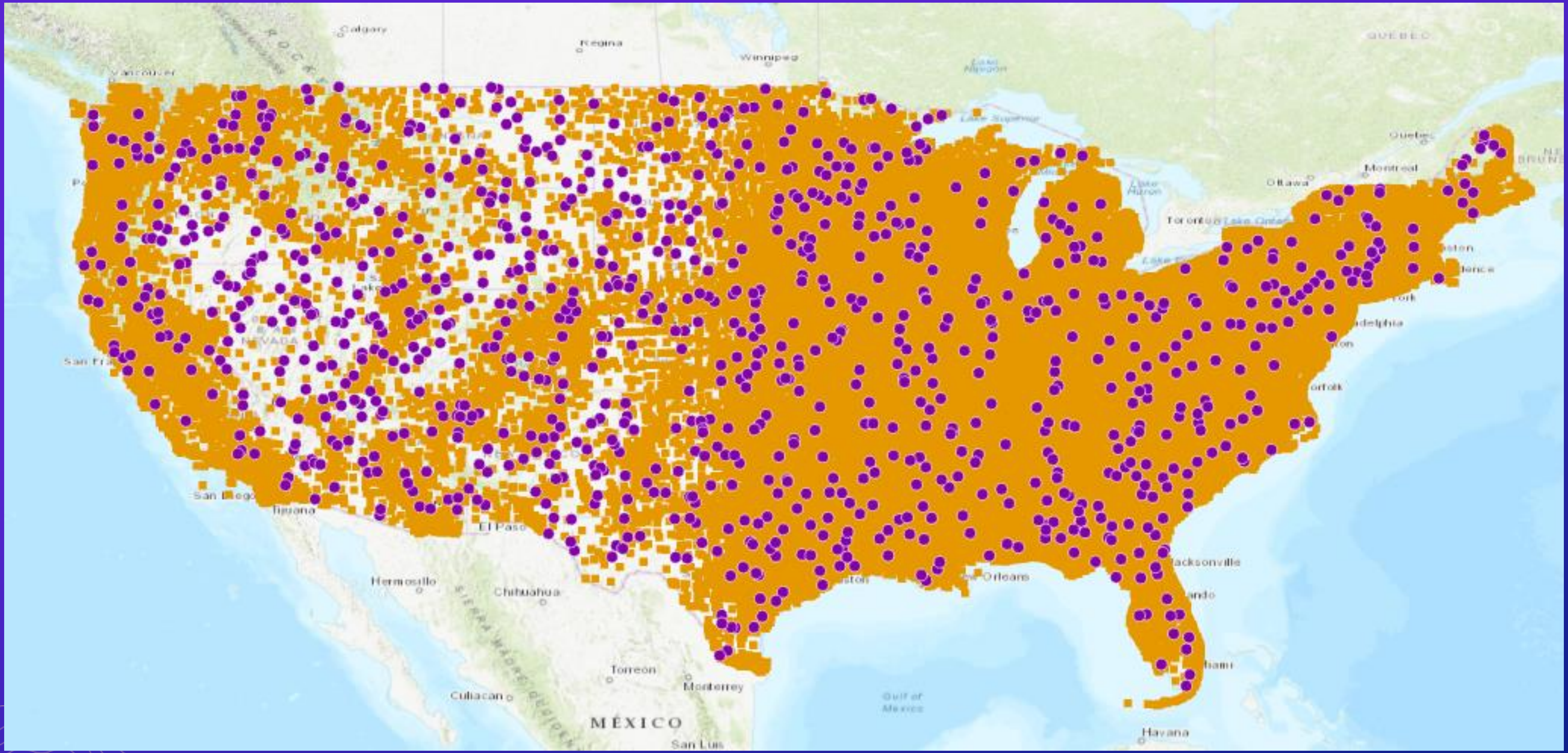
```
# Spatially sort input features
logger.debug("Spatially sorting %s", input_features)
result = arcpy.management.Sort(input_features, output_features,
                               [[desc_input_features.shapeFieldName, "ASCENDING"]], "PEANO")
```

All origins and all destinations

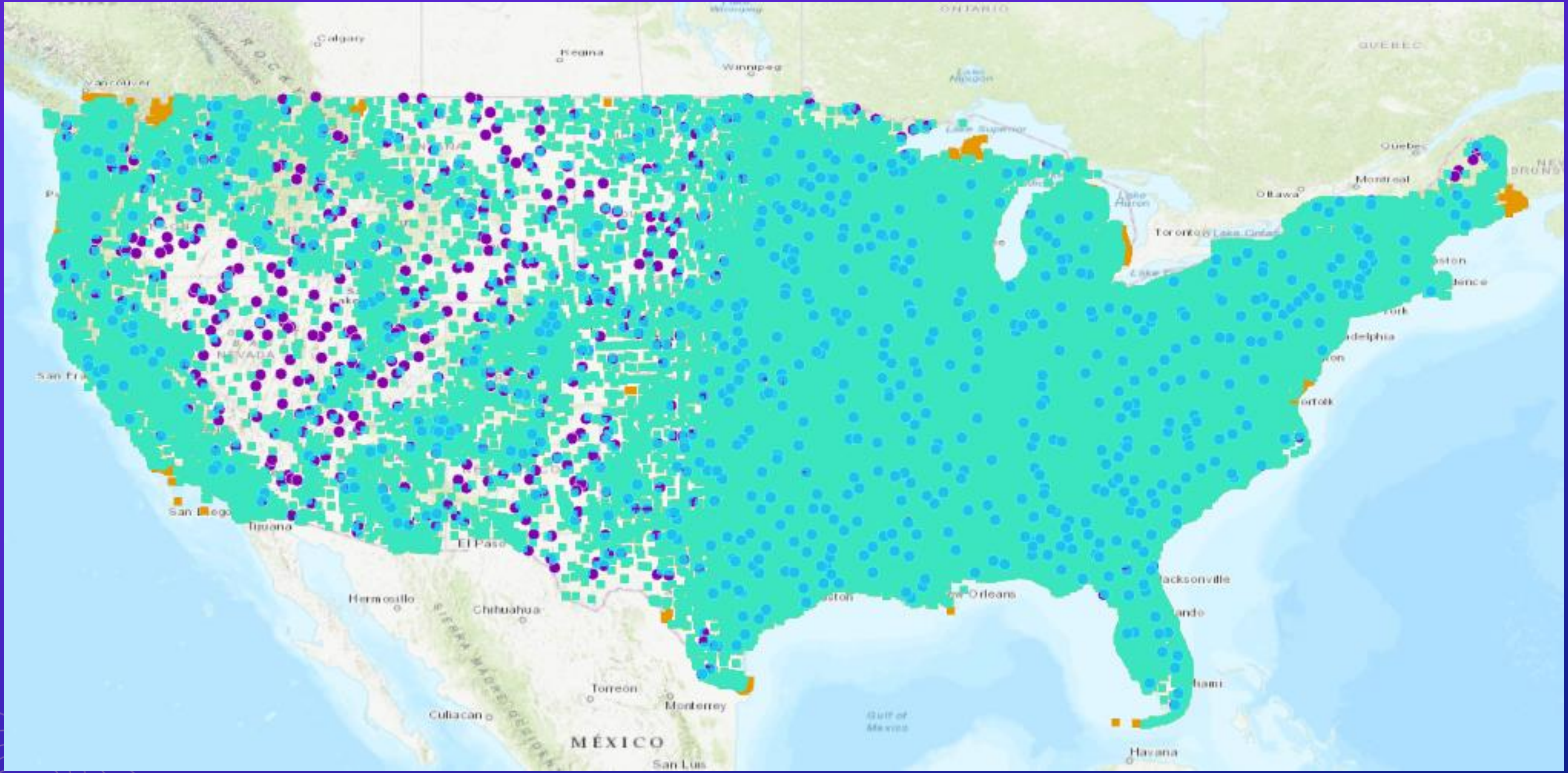


26 million origins ●, 220k destinations ■

Chunk of 1000 unsorted origins

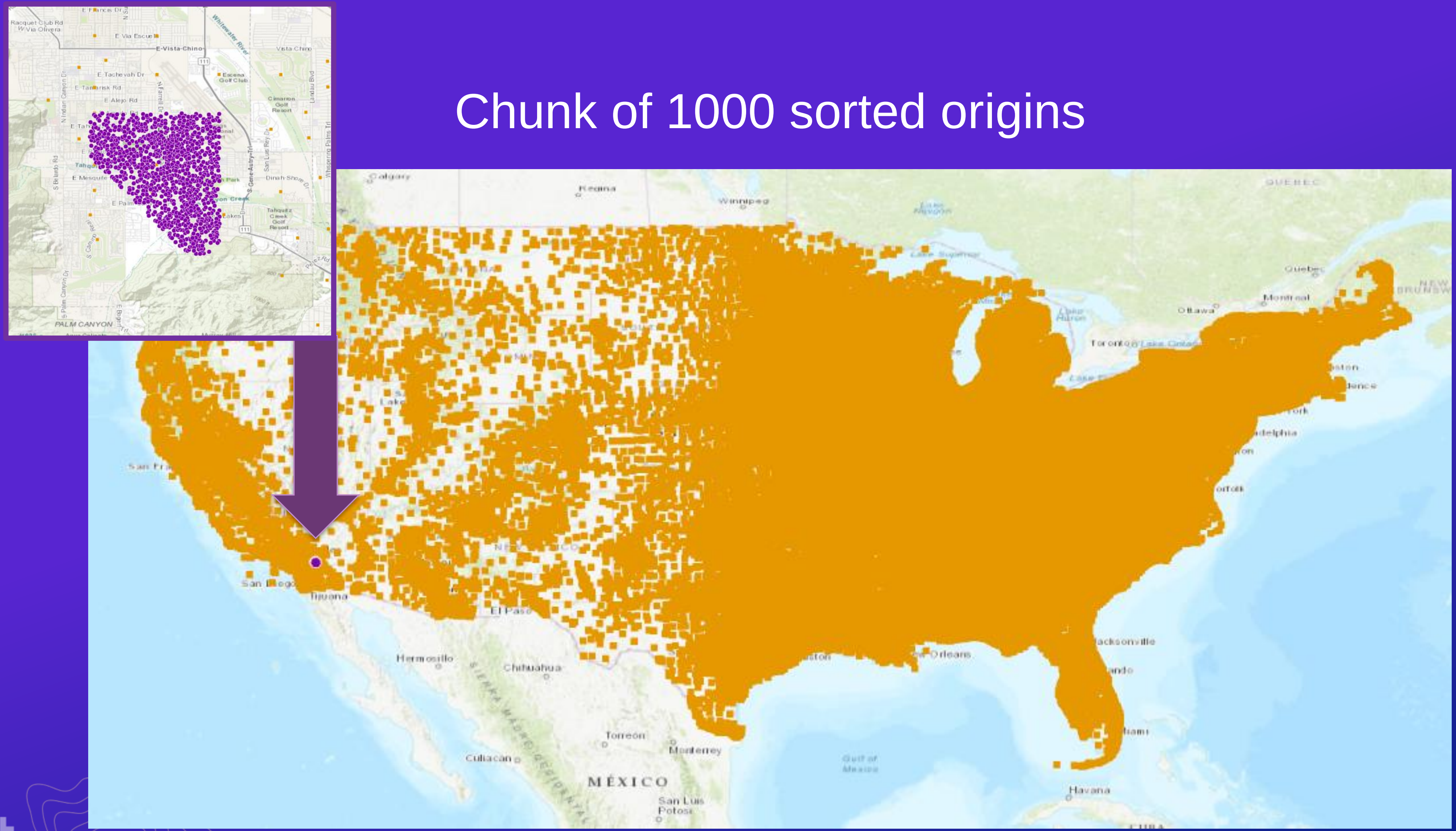


Destinations within 100 miles of 1000 origins

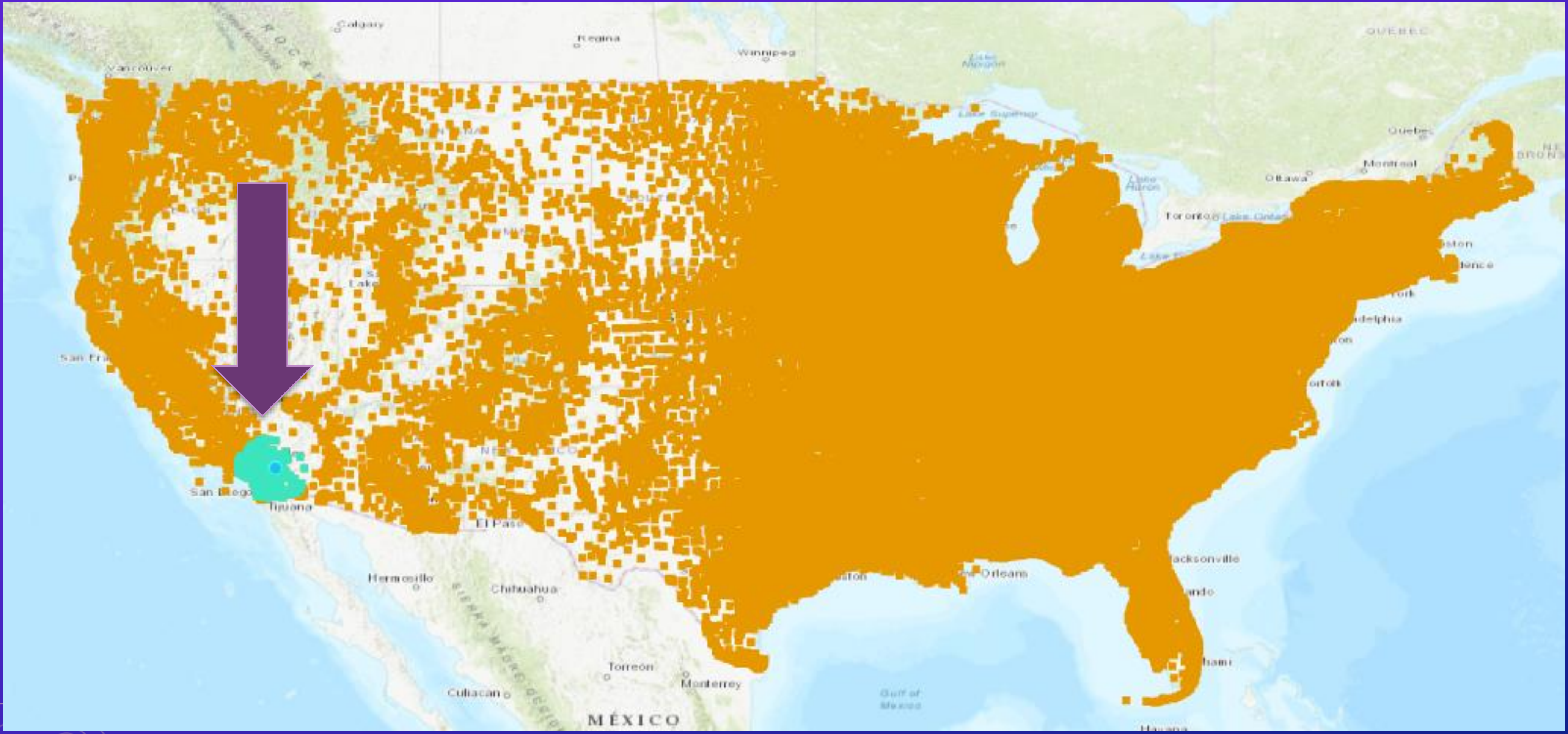


214,449 destinations selected (98%)

Chunk of 1000 sorted origins



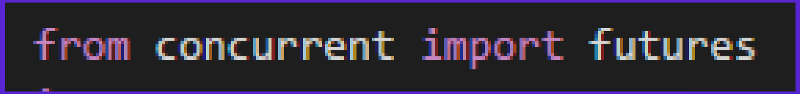
Destinations within 100 miles of 1000 sorted origins



5,653 destinations selected (3%)

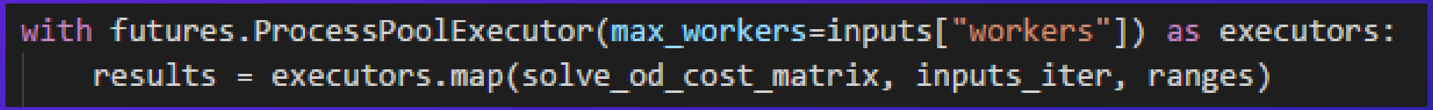
Solve in parallel

- concurrent.futures



```python
from concurrent import futures
```

- Multiprocessing: Spin up multiple processes and run solves on multiple cores

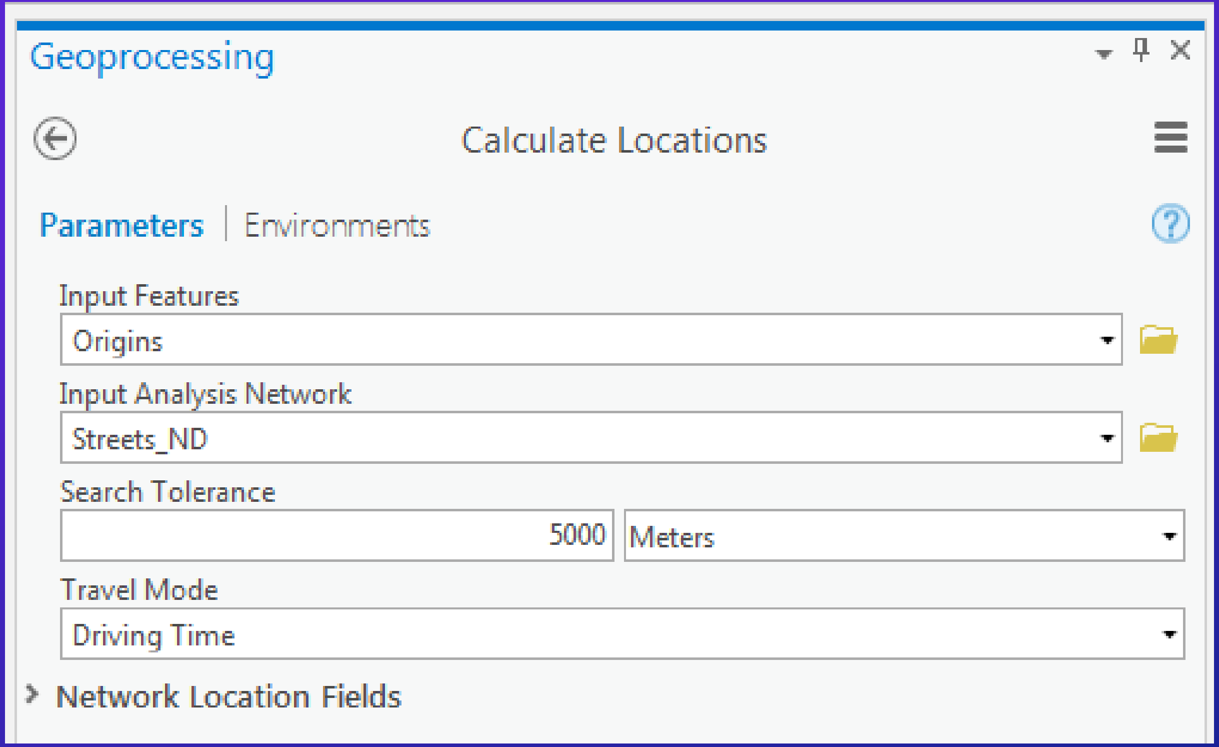


```python
with futures.ProcessPoolExecutor(max_workers=inputs["workers"]) as executors:
    results = executors.map(solve_od_cost_matrix, inputs_iter, ranges)
```

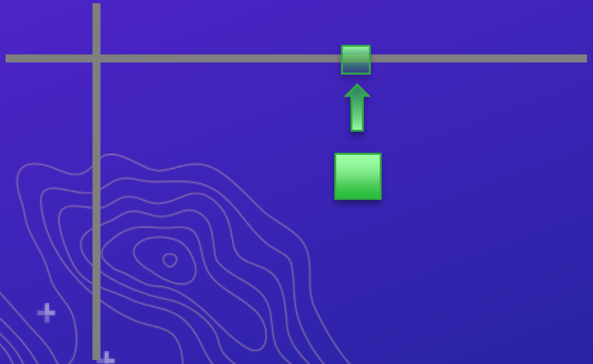
  - Better choice!
  - Works well with arcpy
  - Easy to run from standalone python
    - If running as a geoprocessing script tool, need to use subprocess module
  - Can’t write to same gdb from multiple processes
- Multithreading: Use multiple threads in the same process – DO NOT USE
  - Not good for CPU-intensive problems in python
  - Does not work with arcpy

Pre-calculate location fields

- Only works for local data (not for services)
- Define how a point snaps to the network
- Calculate them in advance if you're using your points more than once
- Use field mapping in Add Locations to use existing location fields



Input Features: Origins
Input Analysis Network: Streets_ND
Search Tolerance: 5000 Meters
Travel Mode: Driving Time
Network Location Fields

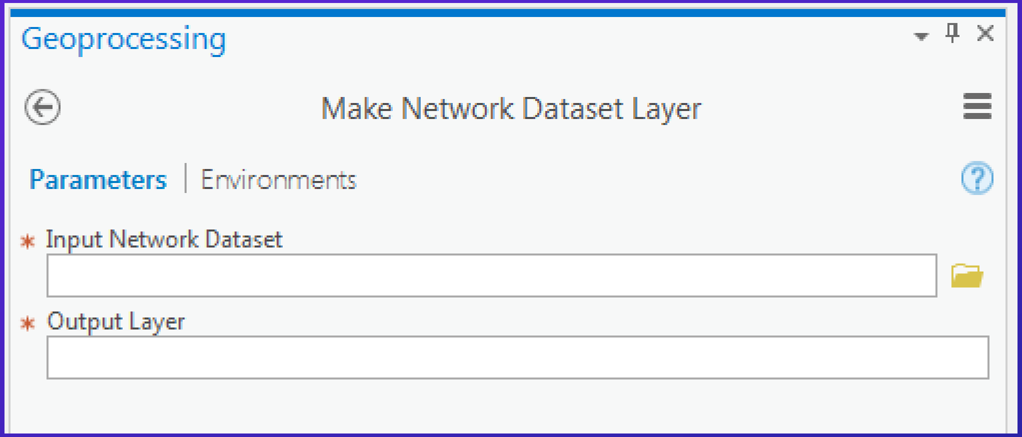


```
logger.debug("Calculating network locations for %s", input_features)
result = arcpy.na.CalculateLocations(output_features, network_data_source, "20 Miles",
                                     ODCostMatrix.get_nds_search_criteria(network_data_source),
                                     travel_mode=travel_mode)
```

SourceID, SourceOID, PosAlong, SideOfEdge

Use network dataset layer

- Opening from catalog path is slow
- Even slower for licensed data or data on UNC path
- Open once by making a Network Dataset Layer; then use the layer name (not the layer object)



```
self.logger.debug("Creating network dataset layer")
arcpy.na.MakeNetworkDatasetLayer(self.network_data_source, self.nds_layer_name)
```

Outline of today's code

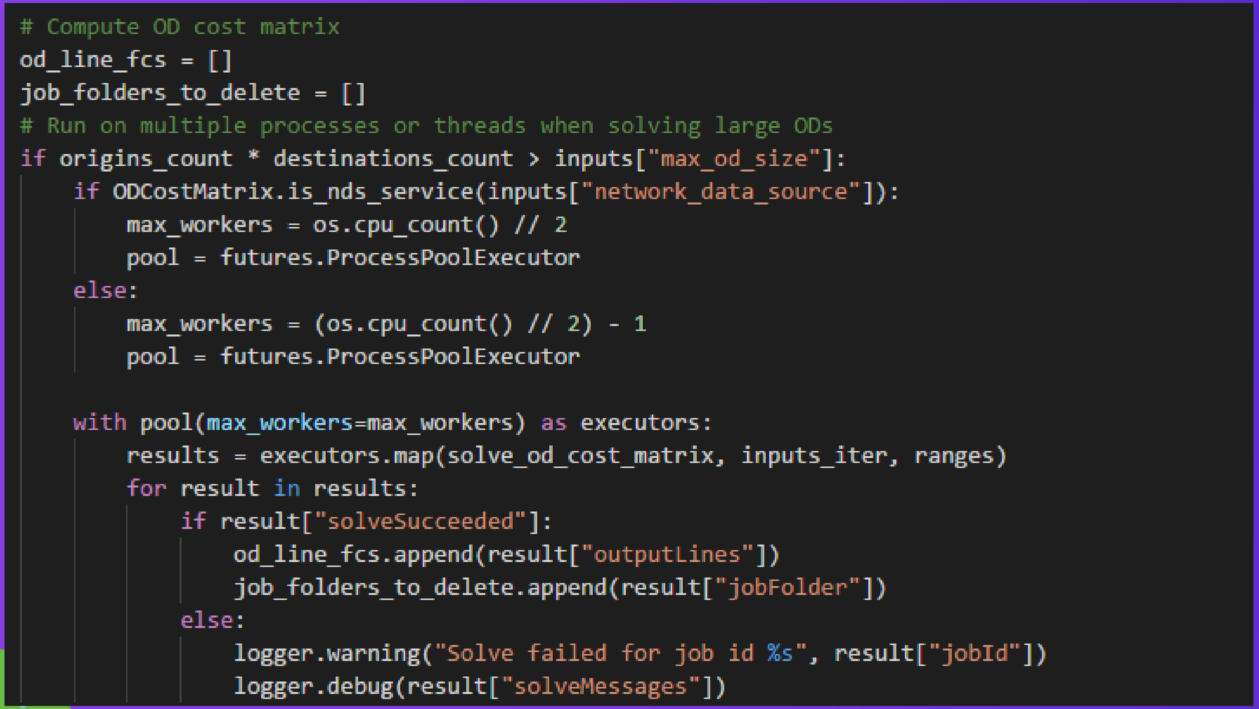
Goal: Solve OD of any size using local data or a service, write it out to a single feature class

- Components:
  - Preprocessing
    - Sorting spatially
    - Calculate network locations
  - Solving
    - Chunking
    - Solving in parallel
  - Post-processing
    - Merging results

Network Analysis Workflow with arcpy.nax

1. Initialize the analysis object (based on a specific network data source)
2. Set the properties for the analysis
3. Load the inputs
4. Solve the analysis
5. Work with the results

Common to all the network analyses



```python
# Compute OD cost matrix
od_line_fcs = []
job_folders_to_delete = []
# Run on multiple processes or threads when solving large ODs
if origins_count * destinations_count > inputs["max_od_size"]:
    if ODCostMatrix.is_nds_service(inputs["network_data_source"]):
        max_workers = os.cpu_count() // 2
        pool = futures.ProcessPoolExecutor
    else:
        max_workers = (os.cpu_count() // 2) - 1
        pool = futures.ProcessPoolExecutor

    with pool(max_workers=max_workers) as executors:
        results = executors.map(solve_od_cost_matrix, inputs_iter, ranges)
        for result in results:
            if result["solveSucceeded"]:
                od_line_fcs.append(result["outputLines"])
                job_folders_to_delete.append(result["jobFolder"])
            else:
                logger.warning("Solve failed for job id %s", result["jobId"])
                logger.debug(result["solveMessages"])
```

Let’s look at some code!

(Local solve)

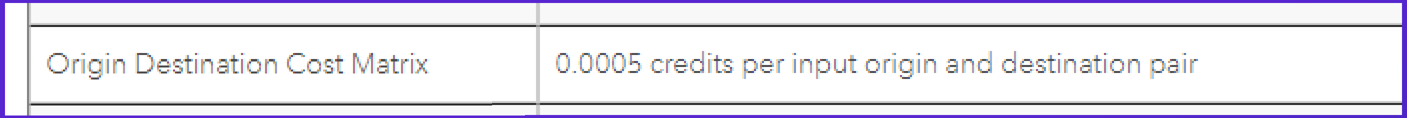
Working with services

Which do I pick?

	Local network dataset	ArcGIS Online service	ArcGIS Enterprise service
Requires your own street data	Yes	No	Yes
Requires Network Analyst extension license	Yes	No	Yes (for Desktop and Server)
Requires service credits	No	Yes	No
Requires ArcGIS Enterprise	No	No	Yes
Number of concurrent processes	Number of CPU cores on your machine	Up to 4	
Analysis limits	Depends on your machine's memory	1000 origins x 1000 destinations	Configurable depending on memory resources of the server

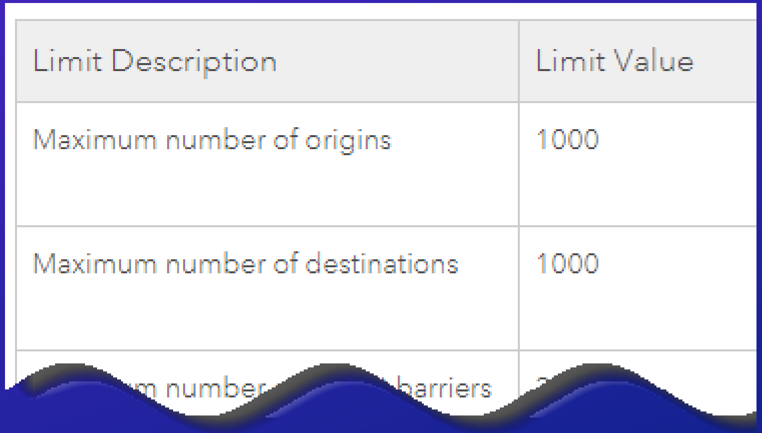
All about credits and service limits!

- Working with ArcGIS Online services requires service credits
  - Credit cost for each service



Origin Destination Cost Matrix	0.0005 credits per input origin and destination pair

- Each service imposes a limit on the maximum problem size that can be solved in one request. For example, OD service only allows 1,000 origins and 1,000 destinations per request.
  - Service limits for each service
    - OD Cost Matrix service
    - Route service
    - Closest Facility service
    - Service Area service
    - Location-Allocation service
    - Vehicle Routing Problem service

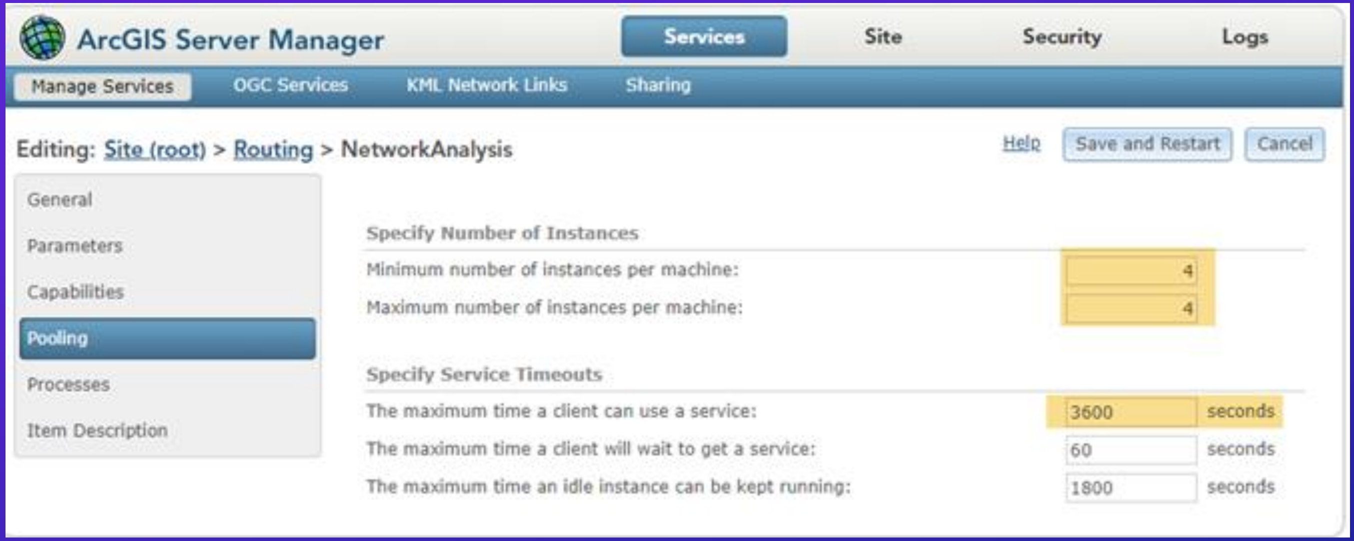


Limit Description	Limit Value
Maximum number of origins	1000
Maximum number of destinations	1000
m number barriers	[illegible]

- Use arcpy.nax.GetWebToolInfo() to retrieve tool limits in a script

How to publish your own routing services to ArcGIS Enterprise

- Use the Publish Routing Services utility
- Special considerations for large problems:
  - Change minimum and maximum service instances to be equal to number of physical CPU cores on your server
  - Set a high value for service usage timeout
  - Set a high value for maximum records returned by server



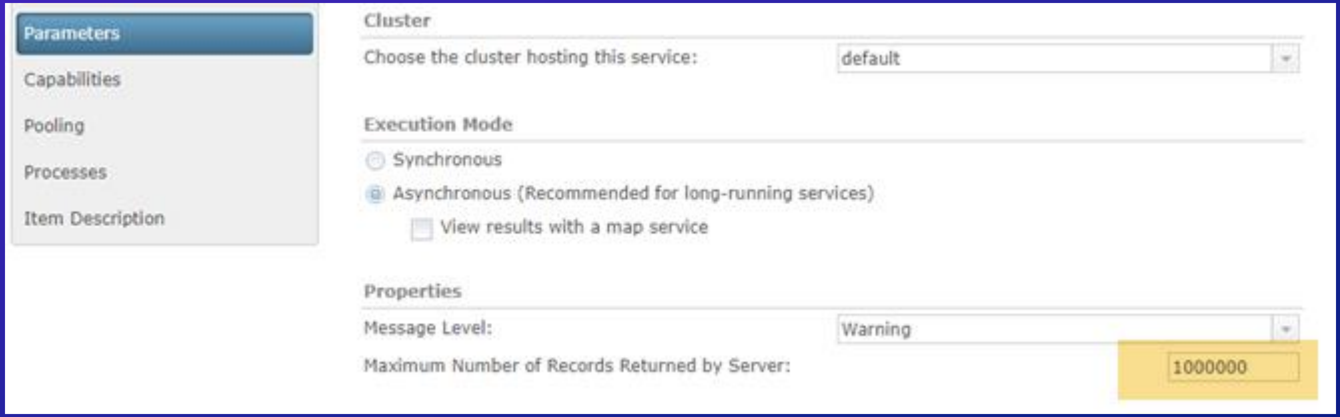
Editing: Site (root) > Routing > NetworkAnalysis

Specify Number of Instances

Minimum number of instances per machine:	4
Maximum number of instances per machine:	4

Specify Service Timeouts

The maximum time a client can use a service:	3600 seconds
The maximum time a client will wait to get a service:	60 seconds
The maximum time an idle instance can be kept running:	1800 seconds



Cluster

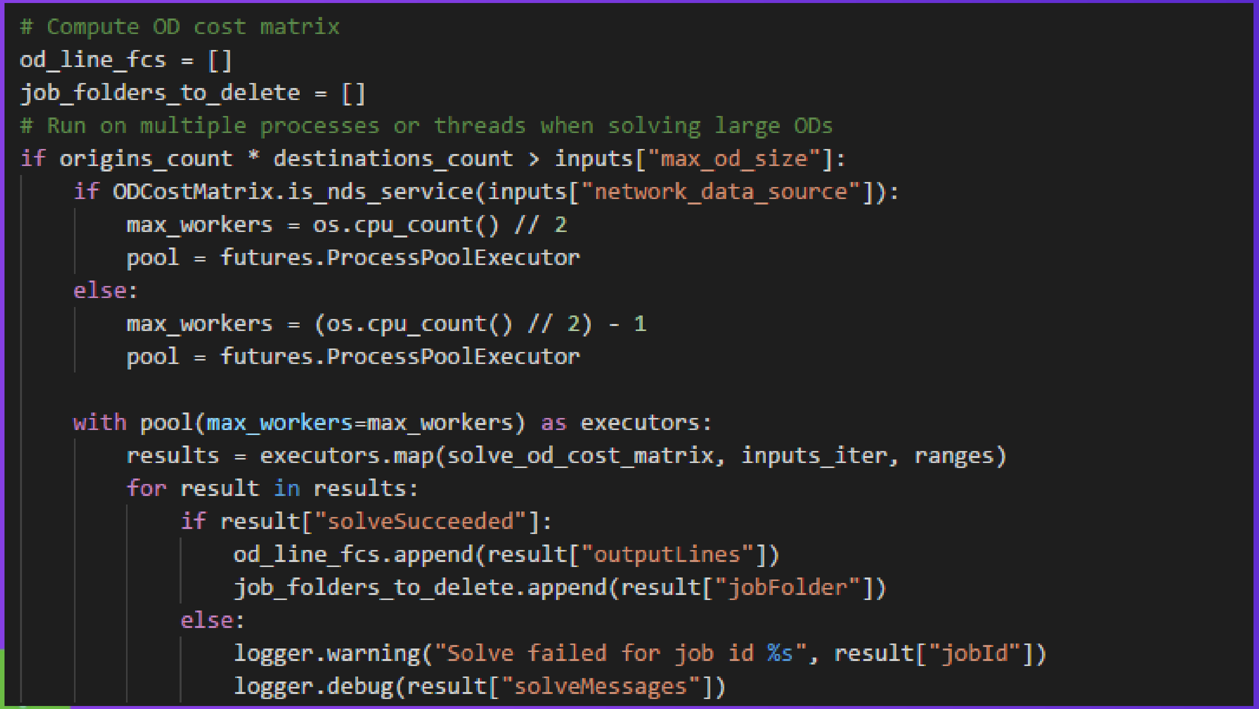
Choose the cluster hosting this service: default

Execution Mode

( ) Synchronous
(●) Asynchronous (Recommended for long-running services)
[ ] View results with a map service

Properties

Message Level:	Warning
Maximum Number of Records Returned by Server:	1000000



```python
# Compute OD cost matrix
od_line_fcs = []
job_folders_to_delete = []
# Run on multiple processes or threads when solving large ODs
if origins_count * destinations_count > inputs["max_od_size"]:
    if ODCostMatrix.is_nds_service(inputs["network_data_source"]):
        max_workers = os.cpu_count() // 2
        pool = futures.ProcessPoolExecutor
    else:
        max_workers = (os.cpu_count() // 2) - 1
        pool = futures.ProcessPoolExecutor

    with pool(max_workers=max_workers) as executors:
        results = executors.map(solve_od_cost_matrix, inputs_iter, ranges)
        for result in results:
            if result["solveSucceeded"]:
                od_line_fcs.append(result["outputLines"])
                job_folders_to_delete.append(result["jobFolder"])
            else:
                logger.warning("Solve failed for job id %s", result["jobId"])
                logger.debug(result["solveMessages"])
```

Let’s look at some code!

(Service solve)

Wrap-up

- Reduce problem size
- Eliminate irrelevant data
- Chunk data
- Spatially sort data
- Solve in parallel
- Pre-calculate network location fields
- Use network dataset layer

Code and slides:

http://esriurl.com/ds20pyslnp