Geospatial Analysis with High Performance Computing Resources

WestGrid Webinar

Ian Percel

University of Calgary, Research Computing Services

December 11, 2019

What is this talk about?

- How do we do spatial analysis without a spatial DataBase like QGIS, PostGRES, or ArcGIS?
- What C and Python libraries do we need in order to perform such an analysis?
- GeoPandas provides data structures and a convenient API for geographic information science
- We can build our own spatial indexes for accelerating spatial joins by using R-Trees [3]
- We will introduce two strategies for parallelizing spatial computations for improved performance

Outline

Outline

Downloading Data 1: csv data

- We will be working with US Census Data from the 5-year American Community Survey
- Specifically, we will be using the de-identified Public Use Microdata Sample (PUMS) data from 2013
- Point your browser at `https://www2.census.gov/programs-surveys/acs/data/pums/2017/5-Year/` to see the relevant FTP directory
- Download `csv_hil.zip` to your personal computer (by right clicking and choosing Save As)

Downloading Data 2: geographies

- Working with PUMS data requires the PUMA boundaries and we will be relating these back to census tracts
- Point your browser at `https://www2.census.gov/geo/tiger/TIGER2018/PUMA/` to see the relevant FTP directory
- Download `tl_2018_17_puma10.zip` to your personal computer (by right clicking and choosing Save As)
- Point your browser at `https://www2.census.gov/geo/tiger/TIGER2018/TRACT/` to see the relevant FTP directory
- Download `tl_2018_17_tract.zip` to your personal computer (by right clicking and choosing Save As)

Where we are going

PUMS Data:

```
import pandas as pd
import numpy as np
from pandas import DataFrame,Series

basedf=pd.read_csv('ss13hil.csv')
#what are the columns?
print(list(basedf.columns))
```

```
['insp', 'RT', 'SERIALNO', 'DIVISION', 'PUMA', 'REGION', 'ST', 'ADJHSG', 'ADJINC', 'WGTP',
'NP', 'TYPE', 'ACR', 'AGS', 'BATH', 'BDSP', 'BLD', 'BUS', 'CONP', 'ELEP', 'FS', 'FULP', 'GASP', 'HFL',
'MHP', 'MRGI', 'MRGP', 'MRGT', 'MRGX', 'REFR', 'RMSP', 'RNTM', 'RNTP', 'RWAT', 'RWATPR', 'SINK', 'SMP',
'STOV', 'TEL', 'TEN', 'TOIL', 'VACS', 'VALP', 'VEH', 'WATP', 'YBL', 'FES', 'FINCP', 'FPARC', 'GRNTP',
'HHL', 'HHT', 'HINCP', 'HUGCL', 'HUPAC', 'HUPAOC', 'HUPARC', 'KIT', 'LNGI', 'MULTG', 'MV', 'NOC', ...]
#plus 50 more real columns and 80 replication weights
```

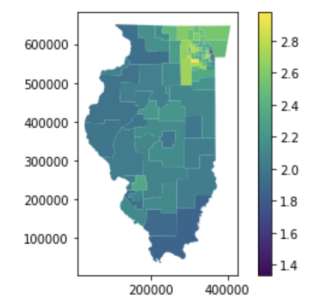
For more information see the pums data dictionary and technical documentation:

https://www2.census.gov/programs-surveys/acs/tech_docs/pums/

Where we are going: geopandas is easy to use for data analysis

What is the mean number of occupants in a census housing unit for each census tract?

```
shp_path='tl_2018_17_puma10.shp'
geo_df=gpd.read_file(shp_path)
housingdf=pd.read_csv('psam_h17.csv', dtype={'PUMA':str})
housingdf['weightedNP']=basedf['WGTP']*basedf['NP']
g = housingdf.groupby(['PUMA'])
pumaAvgNPArray=g['weightedNP'].sum() / g['WGTP'].sum()
avgNPdf=pd.DataFrame(pumaAvgNPArray, columns=['avgNP']).reset_index()
fulldf=geo_df.merge(avgNPdf,how='inner',left_on=['PUMACE10'],right_on=['PUMA'])
fig, ax = plt.subplots(1, 1)
fulldf.plot(column='avgNP', ax=ax, legend=True)
```



Geospatial Computation

Geospatial data analysis is a fairly mature topic computer science and has widely accepted standards for many important algorithms and datasets.

1. Geometric analysis on(to) an ellipsoid (earth measuring)
2. Spatial-relational data analysis (linking data to geometries)
3. Spatial Indexing and search algorithms

These three topics form the core of geospatial data analysis. Other topics, like random process simulation (e.g. MCDS) and modelling of high dimensional geospatial data by stochastic processes (e.g. Kriging) rely on additional data structures and tools that are outside of the scope of this talk. Some of those tools can be found in the PySAL library.

OSGeo Libraries

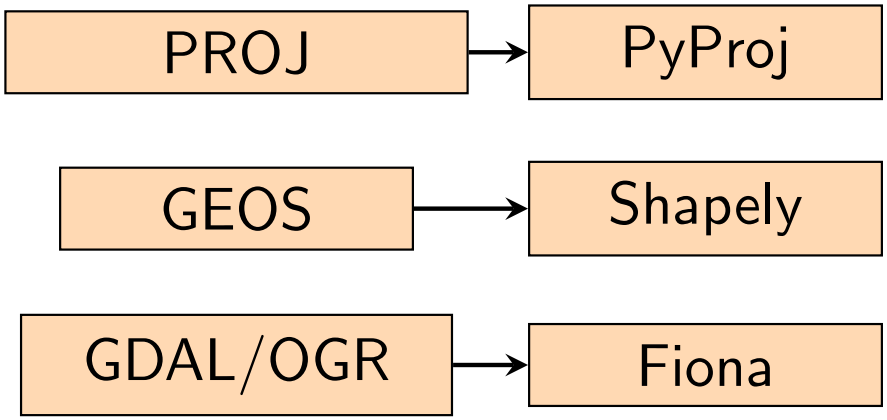
Much as image analysis has been standardized in open source libraries like OpenCV, geospatial analysis algorithms have standard open source implementations as part of the OSGeo projects:

1. PROJ provides generic coordinate transformations for 3D-2D projections
2. GEOS provides tools for computational geometry
3. GDAL/OGR provides geospatial-relational data representation tools

Each of these functionally depends on the previous ones in the list and is an integral part of effective geospatial analysis.

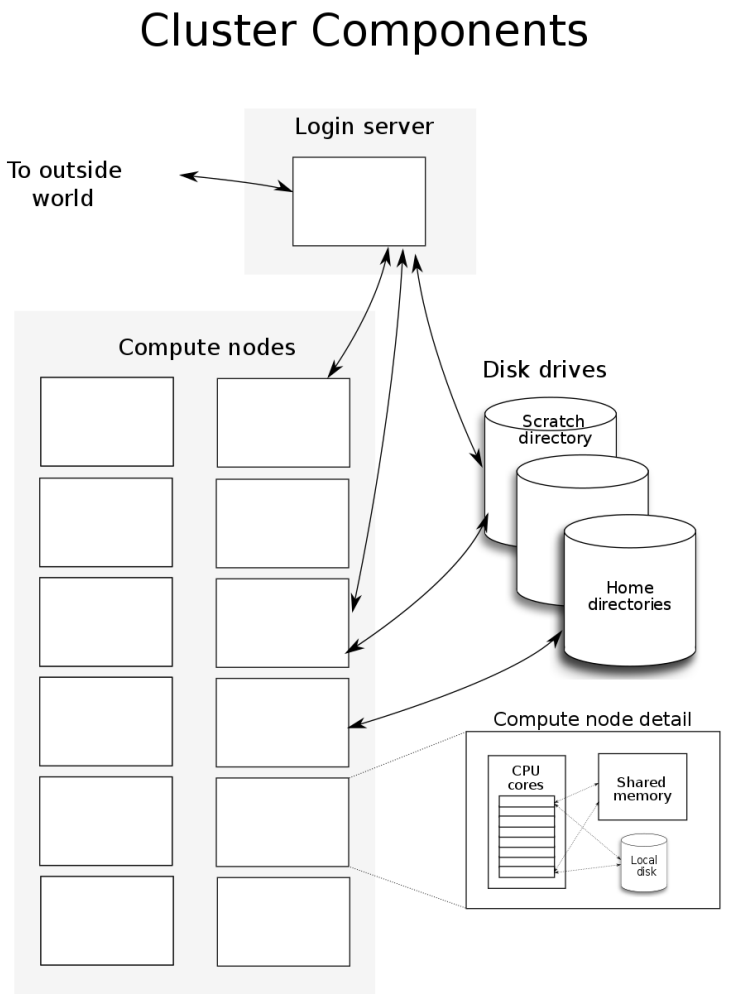
Python Geospatial Libraries

The main libraries provided OSGeo are written in C++. Popular Python modules that have been built on top of them.



These features all come together in a single python library called GeoPandas that mimics the familiar structure of the popular data analysis library Pandas.

Cluster Architecture



Cluster Considerations

What is different about working on an HPC computing cluster from the tradition personal Workstation environment of geospatial analysis?

- Complex C dependencies (like GDAL) need to be built manually (possibly integrated into an environment module by a system administrator)
- Computational resources across many computers can be used on a single job to accelerate calculations
- Managing memory and communication between computers can be a challenge

Python on a Cluster

What is different about working on an HPC computing cluster with Python?

- Python libraries tend to benefit less from this opportunity because of how they manage resources.
- However, if the C libraries that they call (like GDAL) can take advantage of the HPC environment, then the corresponding python tools (like GeoPanadas) can be accelerated.
- Tools like Dask have been built to further simplify using multiprocess or multinode resources at the python level.

What is pandas?

- Pandas provides a SQL-like approach (that blends in elements of statistics and linear algebra) to analyzing tables of data [2]
- DataFrames in R are very similar
- Pandas has been adopted as a de facto standard for input and vectorization across numerous disciplines including Python data analysis with spatial components

Loading data from a csv file

```
basedf=pd.read_csv('ss13hil.csv', index_col='SERIALNO',
          usecols=['SERIALNO', 'PUMA00', 'PUMA10', 'ST',
          'ADJHSG', 'ADJINC', 'WGTP', 'NP', 'TYPE', 'ACR',
          'AGS', 'BATH', 'BDSP', 'BLD', 'BUS', 'CONP', 'ELEP',
          'FS', 'FULP'])
basedf.head()
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000000061	3515	-9	17	1086032	1085467	36	0	1	NaN	NaN	2.0	2.0	8.0	NaN	0.0	NaN	NaN	NaN
2009000000075	1000	-9	17	1086032	1085467	6	1	1	1.0	NaN	1.0	3.0	1.0	2.0	0.0	200.0	2.0	2.0
2009000000108	3402	-9	17	1086032	1085467	15	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	80.0	2.0	2.0
2009000000132	3510	-9	17	1086032	1085467	60	4	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	1.0	2.0	2.0
2009000000150	3518	-9	17	1086032	1085467	37	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	200.0	1.0	2.0

query

- query takes a text string argument in the form (roughly) of a SQL WHERE clause
- Column names need to be referenced without quoting so suitable single-word names are needed
- https://pandas.pydata.org/pandas-docs/version/0.22/indexing.html#indexing-query

```
basedf.query('PUMA00==3515')
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000000061	3515	-9	17	1086032	1085467	36	0	1	NaN	NaN	2.0	2.0	8.0	NaN	0.0	NaN	NaN	NaN
2009000002489	3515	-9	17	1086032	1085467	15	1	1	2.0	1.0	1.0	2.0	2.0	2.0	0.0	50.0	1.0	1.0
2009000002611	3515	-9	17	1086032	1085467	49	2	1	NaN	NaN	1.0	1.0	6.0	NaN	0.0	50.0	2.0	2.0
2009000002724	3515	-9	17	1086032	1085467	45	1	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	40.0	2.0	2.0
2009000006025	3515	-9	17	1086032	1085467	17	2	1	NaN	NaN	1.0	2.0	4.0	NaN	0.0	100.0	2.0	2.0
2009000009853	3515	-9	17	1086032	1085467	14	4	1	1.0	NaN	1.0	4.0	2.0	2.0	0.0	150.0	2.0	2.0
2009000010773	3515	-9	17	1086032	1085467	18	2	1	1.0	NaN	1.0	3.0	3.0	2.0	0.0	110.0	2.0	2.0

merge as JOIN

- merge is a holistic JOIN operator
- Like SQL JOINs, the options for using it are complex and take a great deal of practice to master
- We will focus on two options: on= and how=
- on determines the common column used to join the two together (a list of common columns can be specified)
- note that the indexes are not preserved. To keep them .reset_index() before joining and then set the index from that column after or join on index (not covered here)

```
df1=DataFrame({'a':[1,2,3], 'b':[4,5,6]}, index=['x','y','z'])
df2=DataFrame({'a':[1,2,3], 'c':[10,11,12]}, index=['u','v','w'])
pd.merge(df1,df2,on='a')
```

	a	b	c
0	1	4	10
1	2	5	11
2	3	6	12

map for Transforming Columns

```
basedf['NP\_sq']=basedf['NP'].map(lambda x: x**2)
basedf['PUMA\_str']=basedf['PUMA'].map(lambda x: 'PUMA:'+str(x))
```

Split-Apply-Combine as an overall strategy

- Similar to (but more general than) GROUP BY in SQL
- General tool for bulk changes
- The splitting step breaks data into groups using any column (including the row number) [2]
- This can be accomplished using `df.groupby('colName')`

apply in action

```
subdf=basedf.query('PUMA==03515').copy()
def computeWeightedNP(x):
    x['weightedNP']=x['NP']*x['WGTP']
    return x
subdf=subdf.apply(computeWeightedNP, axis=1)
totals=subdf.sum()
totals['weightedNP']/totals['WGTP']
Out: 1.70737
```

apply as CROSS APPLY

```
def computeWeightedNP(x):
    x['weightedNP']=x['NP']*x['WGTP']
    #print(x)
    totals=x.sum()
    x['avgNP']=totals['weightedNP']/totals['WGTP']
    return x
subdf.groupby(['PUMA']).apply(computeWeightedNP)
```

What's in a GeoDataBase?

- A GeoDB has three essential components: [1]
- Spatial features (with a Datum and Projection information)
- Attributes linked to spatial features
- A means of transforming and linking by attribute data or spatial feature
- Pandas gives us a way of managing structured attribute data, what do we need to add in order to build a usable spatial analysis data structure

GeoDataBases made considerably easier

- GeoPandas supports almost all Pandas operations in one form or another
- GeoPandas provides easy projection handling
- GeoPandas provides R-Tree indexing of GeoDataFrames to accelerate spatial filtering and joining

What is a GeoDataFrame?

- A GeoDataFrame is mostly structured like a DataFrame but has a single column that is a GeoSeries
- This links each attribute record to a unique geospatial feature
- The GeoSeries column can have any name but by default it is `geometry`
- The objects in the geometry column are Shapely objects (in our case Polygons)
- The GeoSeries and GeoDataFrame have a single common `crs` attribute for characterizing the Coordinate Reference System and projection data
- The GeoSeries and GeoDataFrame have a common spatial index attribute `sindex` that implements an R-Tree for the GeoSeries

Loading data to a GeoDataFrame

- Is vastly easier than manually assembling linked spatial data for a Pandas DataFrame
- Automatically identifies the corresponding .prj and .dbf files and incorporates them using `fiona`
- Can still be done manually if something special is needed (`http://geopandas.org/gallery/create_geopandas_from_pandas.html#sphx-glr-gallery-create-geopandas-from-pandas-py`)

```
import geopandas as gpd

shp_path='tl_2018_17_puma10.shp'
geo_df=gpd.read_file(shp_path)
```

Examining our GeoDataFrame

```
geo_df.head()
```

	STATEFP10	PUMACE10	GEOID10	NAMELSAD10	MTFCC10	FUNCSTAT10	ALAND10	AWATER10	INTPTLAT10	INTPTLON10	geometry
0	17	03411	1703411	Cook County (South Central)--Worth & Calumet T...	G6120	S	94129155	1576803	+41.7313634	-087.7167725	POLYGON ((-87.721436 41.734862, -87.721417 41....
1	17	03107	1703107	Will County (Northeast)--Frankfort, Homer & Ne...	G6120	S	243596923	478565	+41.5528956	-087.9168232	POLYGON ((-87.860787 41.557522, -87.857298 41....
2	17	03700	1703700	Kendall & Grundy Counties PUMA	G6120	S	1912412909	37262592	+41.4200983	-088.4339373	POLYGON ((-88.26809799999999 41.724544, -88.26...
3	17	02000	1702000	McLean County PUMA	G6120	S	3064559693	7853695	+40.4945594	-088.8445391	POLYGON ((-88.92933099999999 40.75333699999999...
4	17	01602	1701602	Menard, Logan, De Witt, Piatt, Moultrie, Shelb...	G6120	S	9254202416	88234698	+39.7699432	-089.2258108	POLYGON ((-88.494249 39.215001, -88.4992949999...

```
geo_df.geometry.head()
```

```
0    POLYGON ((-87.721436 41.734862, -87.721417 41....
1    POLYGON ((-87.860787 41.557522, -87.857298 41....
2    POLYGON ((-88.26809799999999 41.724544, -88.26...
3    POLYGON ((-88.92933099999999 40.75333699999999...
4    POLYGON ((-88.494249 39.215001, -88.4992949999...
Name: geometry, dtype: object
```

CRS data and PyProj

- First we need to get a handle on what the `crs` value means and if it agrees with the .prj file provided

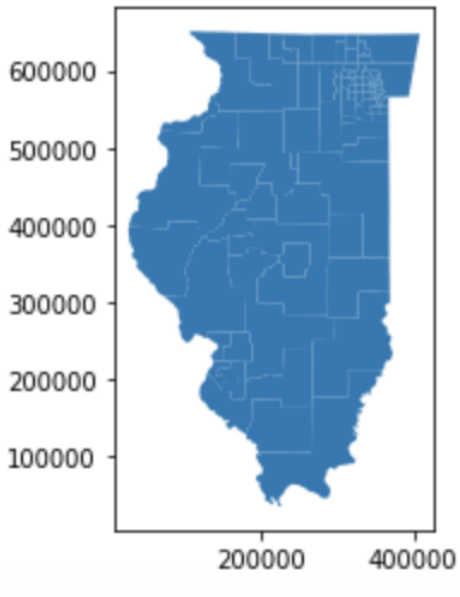
```
from pyproj import CRS

wkt_str='GEOGCS["GCS_North_American_1983",DATUM["D_North_America
crs_utm = CRS.from_string(wkt_str)
crs_utm.to_proj4()
Out: +proj=longlat +datum=NAD83 +no_defs +type=crs
crs_utm.to_epsg()
Out: 4269
```

This establishes that the WKT string from the .prj file has been correctly loaded to the `crs`. We can learn more about the projection in use by looking it up on `https://spatialreference.org/ref/epsg/nad83/` However, we can already tell by examining the proj4 string that the data is unprojected because proj=longlat

Changes of Projection with GeoPandas

- Here we chose a semi-arbitrary projection that works on much of North America but it tailored to the eastern part of Illinois
- Generally care is required in choosing your projection, but the most important thing is consistency
- Differently projected data is fundamentally not comparable

```
geo_df=geo_df.to_crs({'init': 'epsg:26971'})
geo_df.plot()
```



Shapely Polygons

- Full set theoretic machinery: Intersections, Unions, Contains, Differencing
- More geometrically technical options in general but very efficient
- Easy to convert back and forth with PySAL (to get a high level computational geometry interface)

```
import shapely.geometry
poly1=geo_df.geometry[0]
type(poly1)
Out: shapely.geometry.polygon.Polygon
poly2=ps.lib.cg.asShape(poly1)
type(poly2)
Out: pysal.lib.cg.shapes.Polygon
poly3=shapely.geometry.polygon.Polygon(shapely.geometry.asShape(poly2))
type(poly3)
Out: shapely.geometry.polygon.Polygon
poly1==poly3
Out: True
```

Problems 1

What part of Illinois is it that has the lowest accuracy of projection in epsg 26971?

1. Perform the projection of the `geo_df` GeoDataFrame that was outlined in the slides
2. Using the shapely area function through GeoPandas (i.e. `geo_df.geometry.area`) redo the area computation exercise from the last section
3. What is the percent difference in the projected area of each PUMA from the stated land+water areas? What is the maximum observed difference?
4. Use query to find the record with the maximum area difference use the `.plot()` function to plot the PUMA with the biggest error.
5. Use query to plot the PUMAs with a percent error greater than 0.1, greater than 0.08, and greater than 0.05

Solution 1

```
geo_df['statedArea']=geo_df.ALAND10+geo_df.AWATER10
geo_df['computedArea']=geo_df.geometry.area
geo_df['areaDiff']=geo_df['statedArea']-geo_df['computedArea']
geo_df['abs_areaDiff']=geo_df['areaDiff'].abs()
geo_df['frac_areaDiff']=geo_df['abs_areaDiff']/geo_df['statedArea']
geo_df['perc_areaDiff']=geo_df['frac_areaDiff']*100
geo_df['perc_areaDiff'].max()
Out:0.10317568566038449
geo_df.query('perc_areaDiff>0.1')
geo_df.query('perc_areaDiff>0.1').plot()
geo_df.query('perc_areaDiff>0.08').plot()
geo_df.query('perc_areaDiff>0.05').plot()
```

Western Illinois is the worst part of the projection

Problems 2

1. Identify neighboring PUMA regions with a GeoDataFrame (Hint: you can convert the Shapely polygons to PySAL polygons)

Solution 2

```
poly1=ps.lib.cg.asShape(geo_df.geometry[0])
f=lambda poly2: ps.lib.cg.get_shared_segments(poly1,ps.lib.cg.asShape(poly2))
geo_df['sharedSegments']=geo_df.geometry.map(f)

def listFilter(x):
    if x==[]:
        return False
    else:
        return True

geo_df[geo_df['sharedSegments'].map(listFilter)]
```

Extending a GeoDataFrame

- Using the Pandas merge with the spatial frame in the right position returns a DataFrame which would mean giving up our spatial indexing, CRS, and plotting!
- GeoPandas has its own implementations of many standard pandas analysis functions that accept the same options
- Let's start from the reduced DataFrame that was computed earlier using the PUMS weights

```
housingdf['weightedNP']=housingdf['WGTP']*housingdf['NP']
g = housingdf.groupby(['PUMA'])
pumaAvgNPArray=g['weightedNP'].sum() / g['WGTP'].sum()
avgNPdf=pd.DataFrame(pumaAvgNPArray, columns=['avgNP']).reset_index()
```

Extending a GeoDataFrame

- AvgNPdf has the same number of records as our geo_df table because we have made use of groupby
- Join is 1-1 and can be done as an INNER JOIN
- The merge function returns a GeoDataFrame

```
fulldf=geo_df.merge(avgNPdf,how='inner',left_on=['PUMACE10'],right_on=['PUMA'])
type(fulldf)
Out: geopandas.geodataframe.GeoDataFrame
fulldf.head()
```

PUMACE10	GEOID10	NAMELSAD10	MTFCC10	FUNCSTAT10	ALAND10	AWATER10	INTPTLAT10	INTPTLON10	geometry	PUMA	avgNP
03411	1703411	Cook County (South Central)--Worth & Calumet T...	G6120	S	94129155	1576803	+41.7313634	-087.7167725	POLYGON ((350904.9028309903 562835.2311880648,...	03411	2.278784
03107	1703107	Will County (Northeast)--Frankfort, Homer & Ne...	G6120	S	243596923	478565	+41.5528956	-087.9168232	POLYGON ((339419.9439735664 543066.0907343754,...	03107	2.698186
03700	1703700	Kendall & Grundy Counties PUMA	G6120	S	1912412909	37262592	+41.4200983	-088.4339373	POLYGON ((305427.9081081446 561510.3646416094,...	03700	2.696594
02000	1702000	McLean County PUMA	G6120	S	3064559693	7853695	+40.4945594	-088.8445391	POLYGON ((249670.2754834539 453821.0664099103,...	02000	2.203863

GeoDataFrame Queries

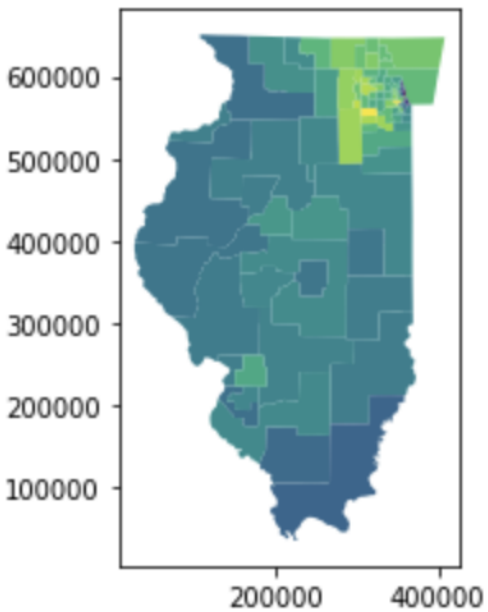
- Joined data can subsequently be filtered as usual

```
fulldf.query('avgNP>2.9')
```

PUMACE10	GEOID10	NAMELSAD10	MTFCC10	FUNCSTAT10	ALAND10	AWATER10	INTPTLAT10	INTPTLON10	geometry	PUMA	avgNP
03106	1703106	Will County (Northwest)--DuPage & Wheatland To...	G6120	S	185174720	2902950	+41.6828015	-088.1450808	(POLYGON ((321438.3752390264 561908.3529506001...	03106	2.965908
03527	1703527	Chicago City (Southwest)--Gage Park, Garfield ...	G6120	S	34025246	48984	+41.7862967	-087.7415184	POLYGON ((351295.6188415109 570459.8131469378,...	03527	2.978911

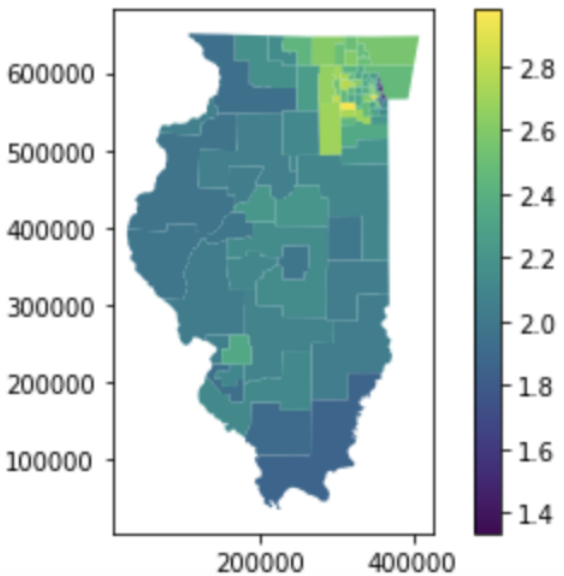
Choropleth Plotting

```
fulldf.plot(column='avgNP')
```



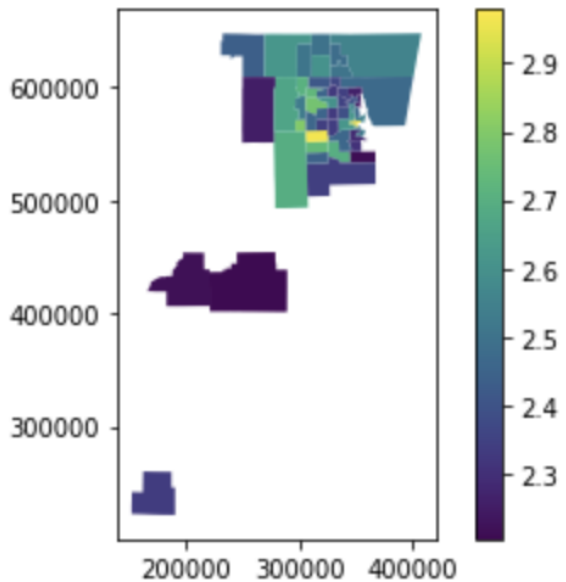
Choropleth Plotting

```
fig, ax = plt.subplots(1, 1)
fulldf.plot(column='avgNP', ax=ax, legend=True)
```



Filtered Choropleth Plotting

```
fig, ax = plt.subplots(1, 1)
fulldf.query('avgNP>2.2').plot(column='avgNP', ax=ax, legend=True)
```



Spatial Index Based Filtering

- Sometimes, we want to analyze explicit spatial subsets
- We could define a mask and test for inclusion row by row
- It is much easier to use the spatial index that already exists

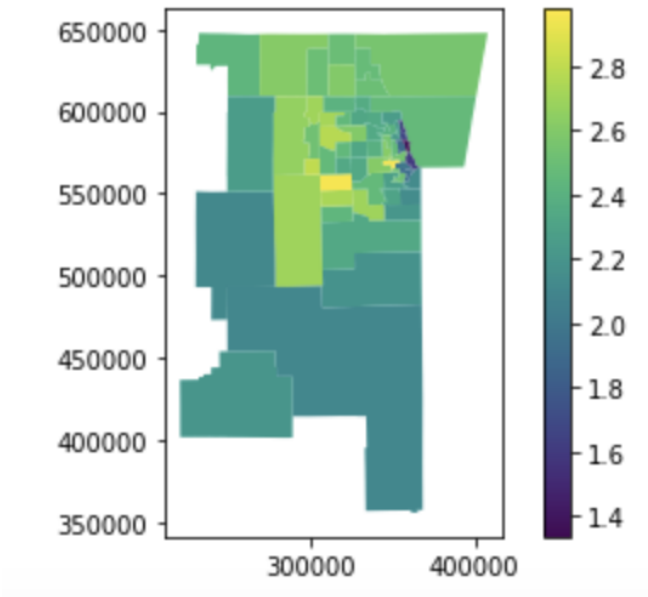
```
fig, ax = plt.subplots(1, 1)
fulldf.cx[250000:,450000:]
```

returns only records with some portion of the polygon east of 250000 and north of 450000 (in the projected coordinate system)

Spatial Index Based Filtering

- Plotting works the same way and allows us to focus our attention on areas of interest

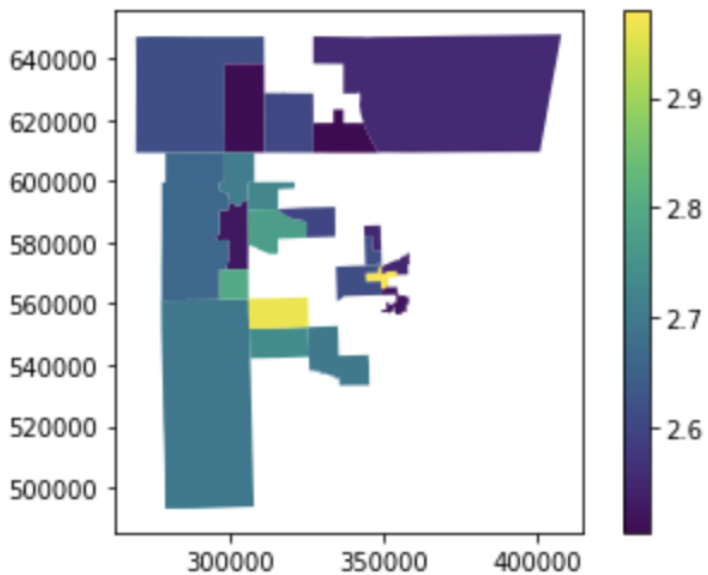
```
fig, ax = plt.subplots(1, 1)
fulldf.cx[250000:,450000:].plot(column='avgNP', ax=ax, legend=True)
```



Spatial Index Based Filtering

- The result can be combined with relational / numerical filtering of values to find records of interest

```
fig, ax = plt.subplots(1, 1)
filteredData=fulldf.cx[250000:,450000:].query('avgNP>2.5')
filteredData.plot(column='avgNP', ax=ax, legend=True)
```



Spatial Joins for linking geographies

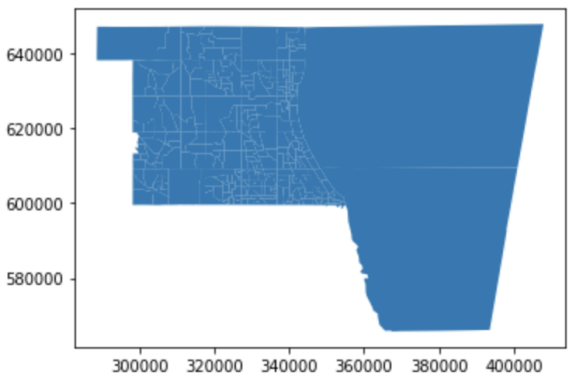
- It is normal to deal with multiple spatial feature sets in geospatial analysis
- Often, different data is attached to each feature and, in order to link data across scales or express connective relationships, it is necessary to perform spatial joins
- Spatial Joins can be thought of as a way of forming a join between two tables of discrete features while using complex spatial relationships as the join criterion rather than using matching keys
- To understand this we will need a second data set that we can join to the first

```
shp_path_t='tl_2018_17_tract.shp'
dft=gpd.read_file(shp_path_t)
dft=dft.to_crs({'init': 'epsg:26971'})
```

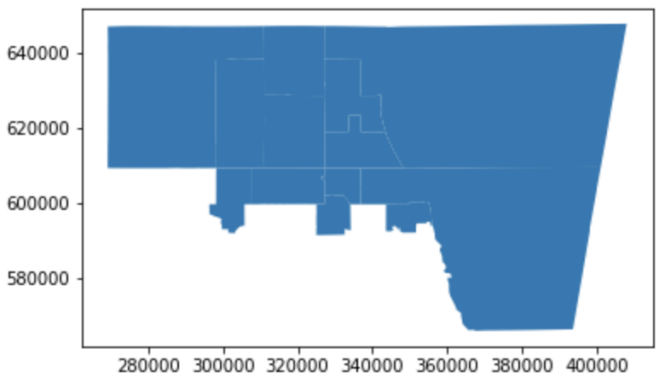
Examining our two geographies

- If two feature sets were the same, comparing them would be uninteresting (or at least very easy)
- It is important to make sure that both are using the same projection

```
dft.cx[300000:,600000:].plot()
```




```
fulldf.cx[300000:,600000:].plot()
```



Adding some simple data to the tract level geography

- In the name of expedience, we will append some randomly generated data to our tract GeoDataFrame

```
import numpy as np
dft['tract_score']=np.random.normal(1000,150,dft.shape[0])
dft.head()
```

TRACTCE	GEOID	NAME	NAMELSAD	MTFCC	FUNCSTAT	ALAND	AWATER	INTPTLAT	INTPTLON	geometry	tract_score
011700	17091011700	117	Census Tract 117	G5020	S	2370100	102060	+41.1294653	-087.8735796	POLYGON ((337416.9310474549 496233.9953135318,...	1253.039315
011800	17091011800	118	Census Tract 118	G5020	S	1790218	55670	+41.1403452	-087.8760059	POLYGON ((336873.9649618285 497112.4753127118,...	892.692410
400951	17119400951	4009.51	Census Tract 4009.51	G5020	S	5170038	169066	+38.7277628	-090.1002620	POLYGON ((145283.7247863071 227488.7024631576,...	724.447989
400952	17119400952	4009.52	Census Tract 4009.52	G5020	S	5751222	305905	+38.7301928	-090.0827510	POLYGON ((146843.1383274837 229402.0122085095,...	1035.095988
950300	17189950300	9503	Census Tract 9503	G5020	S	30383680	349187	+38.3567671	-089.3783135	POLYGON ((205526.7301528867 182696.9930974582,...	1107.998299

How is a spatial join computed and what does that have to do with GEOS/GDAL?

- Spatial join conditions are set relationships between the objects in the geometry column
- In order to determine if two rows "match", the GEOS libraries are called on to evaluate the relationship between the two geometric objects being compared
- For example, two polygons may need to be tested to see if they overlap (i.e. have a nonempty intersection)
- GEOS contains functions for determining the intersection of two polygonal interiors for the purpose of determining intersection and containment (which are the join options supported by GeoPandas)

R-Tree Spatial Indexing

- For large tables of polygons (or other shapes), the intersection determining procedure of GEOS is still not fast enough and a spatial index is required
- As such the `sjoin` function in GeoPandas can be executed either using GEOS or using a spatial index like an R-Tree
- R-Tree is a data structure that (imperfectly) covers the space of possible set operations between polygons,
- This is directly analogous to what a binary search tree does for a key column in a traditional relational database table (or in Pandas for that matter)

rtree library Dependency

- By evaluating set relationships using an R-Tree spatial index, match operations can be accelerated by orders of magnitude
- In order for the RTree version of `sjoin` to work, GeoPandas requires an additional library: `rtree`
- rtree is a wrapper for a C library `libspatialindex`
- As such, we will need to compile libspatialindex and link it to our python distribution

Basic Join Syntax

- `gpd.sjoin(df1,df2,how=, op=)`
- `how` is analogous to how for relational joins except that it also specified which geometry column is retained
- Options: left (df1 geometry is kept and all records from df1), right (df2 geometry is kept and all records from df2), inner (df1 geometry is kept but only matching records from df1)
- `on` is implicit since there is only one GeoSeries per GeoDataFrame
- `op` determines the spatial rule for matching (explanation below for the left and inner cases)
- Options: intersects (any overlap), contains (df1 object entirely surrounds df2 object), within (df2 object entirely surrounds df2 object)

Basic Join Syntax

```
joined_data=gpd.sjoin(tractdf,pumsdf,how='left',op='intersects')
```

Produces a spatially joined GeoDataFrame where the geometry column retained is the tract level geometry and any records associated with PUMAs that it intersects would be appended.

Join Example: Multiple Matches

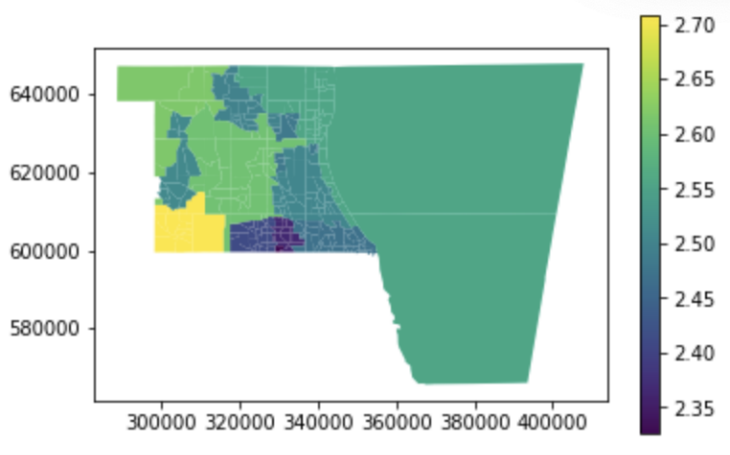
```
joined_data.shape[0]
Out:493
tractdf.shape[0]
Out:287
joined_data.query('TRACTCE=="803500"')
```

	TRACTCE	NAMELSAD	tract_scor	avgNP
1	803500	Census Tract 8035	1158.954562	2.412800
2	803500	Census Tract 8035	1158.954562	2.327185

This result (the joined_data table) has the same number of tracts included as the original filtered tract data, 286.

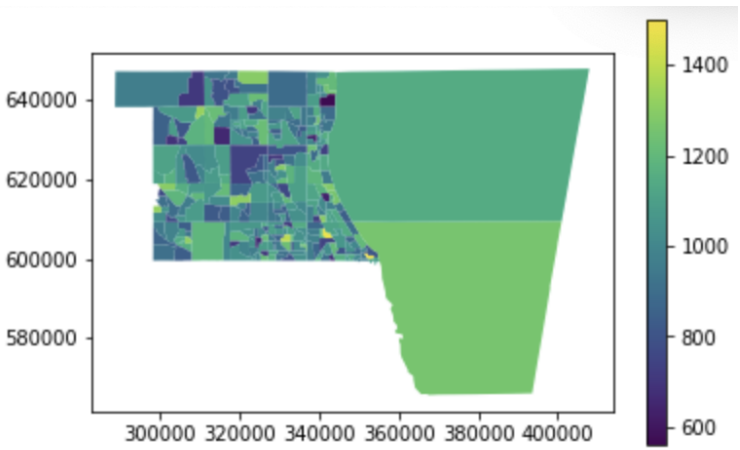
Join Example: Grouped Data

```
def f(x):
    a=x['avgNP'].max()
    y=x.query('avgNP=='+str(a))
    return y

result=joined_data.groupby(['TRACTCE']).apply(f)
fig, ax = plt.subplots(1, 1)
result.plot(column='avgNP', ax=ax, legend=True)
```



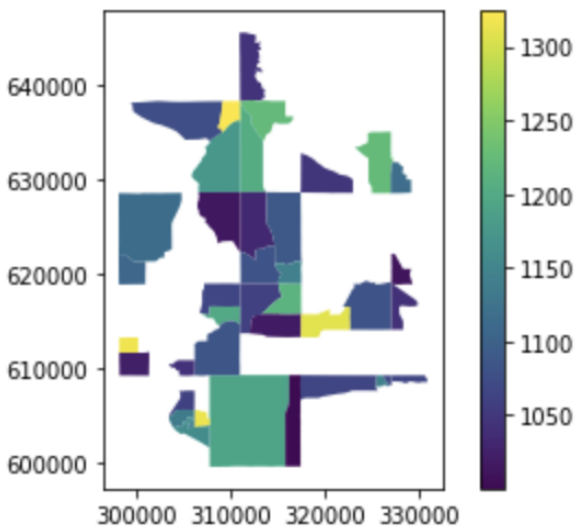
Join Example: Appended Columns

```
fig, ax = plt.subplots(1, 1)
result.plot(column='tract_scor', ax=ax, legend=True)
```



Join Example: Filtering on Joined Data

```
fig, ax = plt.subplots(1, 1)
filt_result=result.query('avgNP>2.6&tract_scor>1000')
filt_result.plot(column='tract_scor', ax=ax, legend=True)
```



Parallelizable Aspects of the Spatial Join Problem

In what follows, we will discuss methods of spatial join acceleration developed for data science applications by
Ravi Shekhar (`https://towardsdatascience.com/geospatial-operations-at-scale-with-dask-and-geopandas-4d9`
Joris Van den Bossche and Mathew Rocklin(`https://matthewrocklin.com/blog/work/2017/09/21/accelerating-geopandas-1`)
These are by no means exhaustive of the options for parallelizing spatial joins. There is a whole body of work on parallelization for the underlying C code. However, these methods are python-centric and can be understood without a deep knowledge of C or MPI.

Parallelizable Aspects of the Spatial Join Problem

- Each test for intersection is independent of all others pairs of polygons being compared
- Multiprocessing versus Multithreading with GDAL/OGR and R-Tree
- RTree Serving
- DataFrame Subclassing

What is Dask?

Dask is a code preprocessing tool for data processing flows that maps a Python algorithm into a graph of tasks.

1. The task graph generated represents each calculation on each data input as a task
2. The task graph is then simplified to something that can run more efficiently in parallel
3. Finally, the graph is submitted to a scheduler that can run the calculation to get answers as they are needed

Dask implements optimized graph simplifications for specific libraries (NumPy and Pandas)

What is a Dask Scheduler?

After generating the task graph, Dask then calls a different Python library to run the parallel calculation as C code. The library used depends on the scheduler that you set when creating the `Cluster` object or calling `compute`.

- threading
- multiprocessing
- dask.distributed $\rightarrow$ mpi4py

mpi4py is the only option that can make effective use of multiple nodes on a shared HPC cluster. In the single node case, there is a choice (in general) between threading and multiprocessing that will be discussed more below.

Dask DataFrames and partitions

The parallelization strategy that we use depends on the objects that we are manipulating.* We will focus on the Dask DataFrame as it is more relevant to the spatial join

- For a spatial join, the primary axis of decomposition is into rows that can each be compared to each row of a target GeoDataFrame
- Dask DataFrames decompose Pandas DataFrames into row batches and distributes them for individual calculations
- Each batch becomes a conceptually independent calculation that we call a partition
- Each partition should have the same function called against it in a given parallel operation

*To handle fast raster computations, we could distribute NumPy `ndarray` objects broken into blocks.

`map_partitions` as an interface for distributed joins

The Dask DataFrame is a viable parallelization strategy for spatial joins as long as one of the tables is a coarser partition (i.e. smaller) than the other.

- To implement the comparison of a subtable join to the target second table we can define a function `custom_join`
- `custom_join` has to operate on a Pandas DataFrame (which will be its first argument) and load a target GeoDataFrame from a file for the second dataset*
- Because the partitioning of Dask DataFrame is optimized for a Pandas DataFrame, we need to use a representation of the source GeoDataFrame as a Pandas DataFrame with the geometry coded as a sequence of text or numeric columns

*functional interface and data passing strategy due to Ravi Shekhar

`map_partitions` as an interface for distributed joins

- The `custom_join` function would take a dataframe as its first argument and then a sequence of column references (so that the source dataframe is arbitrary up to the geometry serialization (in our simple case we will use the name of a single geometry column)
- Finally, when executing the computation in Dask we would call `sourceGDF.map_partitions(custom_join, col1, col2, ... , meta=(outputCol,np.float64))` and it would execute the function on each partition
- The meta tuple is used to fix a name and datatype for the output column instead of leaving it up to Dask to infer. The advantage of `np.float64` is that is permits NaNs for unmatched rows.

Choosing a Dask Scheduler

- GDAL/OGR and libspatialindex are NOT threadsafe therefore the threading engine is not an option
- `threading` is the default option for Dask DataFrame because Pandas DataFrames themselves regularly release the GIL and threadsafe
- Therefore, to make sure that you handle data in a way that is consistent you need to generate a new process for each calculation

Choosing a Dask Scheduler

- `mpi4py` and `multiprocessing` both produce new processes (and an independent python interpreter) for each partition of the data and as such can be used safely
- This does mean that RTrees and target GeoDataFrames need to be built separately in each worker process increasing the memory overhead (unless an explicit shared memory construct were carefully invoked)
- On a single node, `multiprocessing` is probably the best choice as it allows for separate core to being work on separate join components safely as long as you have enough memory
- Across multiple nodes, `dask.distributed` supported by `mpi4py` is the only practical choice

Practical Implementation

We begin by converting the GeoDataFrame to a Pandas DataFrame. One approach by which this can be done is simply using the DataFrame constructor to make the `geometry` column into a column of Shapely objects (a valid column in a Pandas DataFrame) rather than a GeoSeries (which includes additional metadata and indexing data structures). We will focus on the example join above of tractdf and pumsdf

```
tdf_pandas=pd.DataFrame(tractdf)
tdf_dask=dd.from_pandas(tdf_panadas, npartitions=8)
```

Practical Implementation

The resulting dask DataFrame has been allocated a number of partitions such that one partition per worker can be employed and have a dedicated core for each worker on an 8 core system. To scale this up we could simply use a system with more cores and more partitions. Note that we are partitioning the tract DataFrame because it has finer spatial partitions and so more rows. For a larger DataFrame, we could use 40 cores (in our sbatch request) and 40 partitions.

```
#tractdf should be a suitably projected GeoDataFrame
tdf_pandas=pd.DataFrame(tractdf)
tdf_dask=dd.from_pandas(tdf_panadas, npartitions=40)
```

Practical Implementation

With a partitioned dataframe in hand, we can define a join function

```
def custom_join(df, geometry, shp_path):
    import geopandas as gpd
    import pandas as pd
    #read in PUMA df as GeoPandas DF
    pumaGDF=gpd.read_file(shp_path)
    pumaGDF.to_crs({'init': 'epsg:26971'})
    #convert df (partition of tract df) back to geopandas DF
    tractGDF=gpd.GeoDataFrame(df,geometry=geometry)
    #with RTrees built for each, execute usual sjoin
    joined_data=gpd.sjoin(tractGDF,pumaGDF,how='left',op='intersects')
    #convert joined DF back to DaskDF
    sjoinOutputDask=pd.DataFrame(joined_data)
    return sjoinOutputDask
```

Practical Implementation

Finally, we can execute the code using the `map_partitions` Dask function

```
#define an empty dataframe with appropriate column names
#and datatypes for the join output
EmptyDF=...
with dask.config.set(scheduler='processes'):
    tdf_dask.map_partitions(custom_join, 'geometry' ,'aPath', meta=EmptyDF)
    joined_data=gpd.GeoDataFrame(tdf_dask,geometry='geometry')
```

Practical Implementation

Improvements

- We can reduce memory overhead between processes by only loading geopandas and pandas initially in the outer process
- We can also reduce memory overhead by creating shared memory construct for sharing out the target dataframe (the one not being partitioned) to the worker processes
- We can even build a central RTree server and share out the target RTree to minimize records shared between processes (described in the next slide)
- All of the converting back and forth is wasteful. We are better off subclassing Dask DataFrame directly `https://docs.dask.org/en/latest/dataframe-extend.html` and having partitions operate directly on sub GeoDataFrames

Practical Implementation

- None of this work makes much sense without a very large input data set.
- RTrees are already very efficient search mechanisms.
- Each partition should be on the order of at least 1 GB, per usual Dask chunksize recommendations for workers.

A Different Approach to Interprocess Communication

- Alternatively one can generate one RTree for each GeoDataFrame and then have interprocess communication in the form of an RTree server using multiprocessing manager to share out the relevant RTree filtered joins to the worker processes
- `https://sgillies.net/2008/10/30/multiprocessing-with-rtree.html`
- This requires a substantial amount of additional work to make sure that the memory access is done in a safe way without introducing too much overhead
- The key idea is register needed geometric operations with a multiprocessing manager subclass and then call them from the remote workers

[1] Paul Bolstad. *GIS Fundamentals*. University of Minnesota, 2012.

[2] Wes McKinney. *Python for Data Analysis*. O'Reilly, 2013.

[3] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.