Grid Coordinate Conversion with Dask cuDF

In this notebook you will extend your understanding and ability to work with Dask cuDF by revisiting the user-defined grid conversion function. In doing so you will learn more about how Dask distributes the work of computational graphs and will continue preparing data for GPU-accelerated machine learning in the next section of the workshop.

Objectives

By the time you complete this notebook you will be able to:

- Use Dask cuDF to map user-defined functions over Dask cuDF dataframe partitions

Imports

We create a Dask cluster before importing `dask_cudf` to ensure the latter has the right CUDA context. We will import the elements necessary for creating the Dask cluster and wait to import `dask_cudf` until after the cluster has been created.

In [1]:
```
import subprocess

from dask.distributed import Client, wait, progress
from dask_cuda import LocalCUDACluster

import dask.dataframe as dd
```

In [2]:
```
cmd = "hostname --all-ip-addresses"
process = subprocess.Popen(cmd.split(), stdout=subprocess.PIPE)
output, error = process.communicate()
IPADDR = str(output.decode()).split()[0]

cluster = LocalCUDACluster(ip=IPADDR)
client = Client(cluster)
client
```

```
distributed.preloading - INFO - Import preload module: dask_cuda.initialize
distributed.preloading - INFO - Import preload module: dask_cuda.initialize
distributed.preloading - INFO - Import preload module: dask_cuda.initialize
distributed.preloading - INFO - Import preload module: dask_cuda.initialize
```

Out[2]:

Client

Client-bc387b2d-841a-11ee-845d-0242ac120003

Connection method: Cluster object **Cluster type:** dask_cuda.LocalCUDACluster

Dashboard: http://172.18.0.3:8787/status

▸ Cluster Info

Now we Import CUDA context creators after setting up the cluster so they don't lock to a single device.

In [3]:

```
import cudf
import dask_cudf

import cupy as cp
```

Lat/Long to Grid Coordinate Conversion with Dask

We return again to converting latitude and longitude coordinates into grid coordinates by applying our custom `latlong2osgbgrid` function, however this time we will do so in a distributed fashion with Dask. Before we can do so, we need to discuss a little more specifically about how Dask distributes the computation of its task graphs.

Dask Partitions

Internally, Dask dataframes are split into a number of partitions, each being an individual cuDF dataframe. Under the hood, Dask automatically breaks up the work of dataframe methods and operations among these partitions, taking care to communicate efficiently and correctly. For this reason, in using Dask earlier today to perform Dask dataframe operations, you did not have to think explicitly about how Dask had partitioned the Dask dataframes.

However, when we would like to work with Dask dataframes outside their built-in methods and operators, such as when applying custom functions, we often need to work more explicitly with the partitions of the Dask dataframe, as we will do now.

Dask Grid Converter

Ultimately, we are going to map our custom function to each partition of a Dask dataframe using the dataframe's `map_partitions` method.

With this in mind, let's look at `latlong2osgbgrid_dask` , noting modifications we have had to make to its CuPy counterpart in order to work effectively when mapped to Dask dataframe partitions rather than run on cuDF columns. There are 4 parts to the refactor, each with accompanying comments.

In [4]:
```
# 1) Rather than passing in `lat` and `long` arguments, we pass in a dataframe partiti
def latlong2osgbgrid_dask(part_df, lat_col='lat', long_col='long', input_degrees=True)
    '''
    Converts latitude and longitude (ellipsoidal) coordinates into northing and eastir

    Inputs:
    part_df: the dask distributed dataframe partition
    lat_col: the name of the column holding latitude data
    long_col: the name of the column holding longitude data
    input_degrees: if True (default), interprets the coordinates as degrees; otherwise

    Output:
    original dataframe with northing and easting columns concatenated to the right
    '''

    # 2) Our previous function expected `lat` and `long` values to each be CuPy array-
    lat = cp.asarray(part_df[lat_col])
    long = cp.asarray(part_df[long_col])

    # 3) At this point we reuse the previous cupy code until it is time to return valu
    if input_degrees:
        lat = lat * cp.pi/180
        long = long * cp.pi/180

    a = 6377563.396
    b = 6356256.909
    e2 = (a**2 - b**2) / a**2

    N0 = -100000 # northing of true origin
    E0 = 400000 # easting of true origin
    F0 = .9996012717 # scale factor on central meridian
    phi0 = 49 * cp.pi / 180 # latitude of true origin
    lambda0 = -2 * cp.pi / 180 # longitude of true origin and central meridian

    sinlat = cp.sin(lat)
    coslat = cp.cos(lat)
    tanlat = cp.tan(lat)

    latdiff = lat-phi0
    longdiff = long-lambda0

    n = (a-b) / (a+b)
    nu = a * F0 * (1 - e2 * sinlat ** 2) ** -.5
    rho = a * F0 * (1 - e2) * (1 - e2 * sinlat ** 2) ** -1.5
    eta2 = nu / rho - 1
    M = b * F0 * ((1 + n + 5/4 * (n**2 + n**3)) * latdiff -
                  (3*(n+n**2) + 21/8 * n**3) * cp.sin(latdiff) * cp.cos(lat+phi0) +
                  15/8 * (n**2 + n**3) * cp.sin(2*(latdiff)) * cp.cos(2*(lat+phi0)) -
                  35/24 * n**3 * cp.sin(3*(latdiff)) * cp.cos(3*(lat+phi0)))
    I = M + N0
    II = nu/2 * sinlat * coslat
    III = nu/24 * sinlat * coslat ** 3 * (5 - tanlat ** 2 + 9 * eta2)
    IIIA = nu/720 * sinlat * coslat ** 5 * (61-58 * tanlat**2 + tanlat**4)
    IV = nu * coslat
    V = nu / 6 * coslat**3 * (nu/rho - cp.tan(lat)**2)
    VI = nu / 120 * coslat ** 5 * (5 - 18 * tanlat**2 + tanlat**4 + 14 * eta2 - 58 * t

    northing = I + II * longdiff**2 + III * longdiff**4 + IIIA * longdiff**6
    easting = E0 + IV * longdiff + V * longdiff**3 + VI * longdiff**5
```

```
    # 4) Having calculated `northing` and `easting`, we add them as series to our part
    part_df['northing'] = cudf.Series(northing)
    part_df['easting'] = cudf.Series(easting)

    return(part_df)
```

Mapping Functions to Partitions

The Dask dataframe `map_partitions` method applies a given function to each partition. As you saw in the `latlong2osgbgrid_dask` function, at least one of the arguments to the function should be a `dask.dataframe` (in our case, `part_df`).

The other requirement for `map_partitions` is a *meta*: a dataframe with the structure that we will be returning from the function. You can think of this like defining a function signature, and in fact, you will find many instances in Dask programming where a meta is required.

In our case, however, Dask can automatically infer the meta from our function and its inputs, so we don't need to provide one explicitly.

Using the Parquet Format

The csv format we have using thus far has been realistic to many data scientists' experiences, but alternatives are often more efficient for our needs.

Here, we will output to the columnar Apache Parquet format, a natural companion to the Apache Arrow memory format of RAPIDS. Parquet also will compress our data from about 18Gb to about 12Gb.

The `to_parquet` writer will create a folder of smaller parquet files with associated metadata that can efficiently be read back in later with `read_parquet` , taking advantage of parallel I/O with multiple GPU workers in Dask.

Exercise: Build a Dask Grid Converter Pipeline

You can now build a simple data pipeline to add OSGB36 grid coordinates to the population data set. This will consist of three steps:

1. Read the csv file at `./data/pop5x_1-07.csv` into a Dask dataframe with `read_csv`
2. Map the function `latlong2osgbgrid_dask` over that dataframe with `map_partitions`
3. Write the results to the parquet format in the folder `pop5x` with `to_parquet`

While this is running, consider bringing up the Dask status dashboard on port 8787, as in the previous notebook, and observe how Dask is asynchronously reading, transforming, and writing data.

In [6]:
```
ddf = dask_cudf.read_csv('./data/pop5x_1-07.csv')
ddf = ddf.map_partitions(latlong2osgbgrid_dask)
ddf.to_parquet('pop5x')
```

Out[6]:
```
[None]
```

Solution

In [7]:
```
# %load solutions/csv_to_parquet_pipeline
ddf = dask_cudf.read_csv('./data/pop5x_1-07.csv')
ddf = ddf.map_partitions(latlong2osgbgrid_dask)
ddf.to_parquet('pop5x')
```

Out[7]:
```
[None]
```

Exercise: Compute Grid Coordinate Statistics

You can analyze the results of mapping `latlong2osgbgrid_dask` the same way as any other `dask_cudf` dataframe columns. We can also see the speed enabled by parquet in the following two steps:

1. Read the `pop5x` folder of parquet files into a Dask dataframe
2. Compute the mean of the `northing` and `easting` columns

Observe how quickly Dask can read in the 12Gb of parquet files through this method.

In [9]:
```
ddf = dask_cudf.read_parquet('pop5x')
ddf[['northing', 'easting']].mean().compute()
```

Out[9]:
```
northing    273564.672821
easting     447839.348862
dtype: float64
```

Solution

In [10]:
```
# %load solutions/read_parquet
ddf = dask_cudf.read_parquet('pop5x')
ddf[['northing', 'easting']].mean().compute()
```

Out[10]:
```
northing    273564.672821
easting     447839.348862
dtype: float64
```

Next

This concludes the first section of the workshop. You've already learned how to use cuDF and Dask_cuDF to explore and modify data, including data sets larger than a single GPU's memory, and have successfully prepped several data sets for GPU-accelerated machine learning.

In the next section of the workshop, you will use the data you have prepped in the context of several GPU-accelerated machine learning algorithms, before moving onto the final section of the workshop where you will apply both your GPU-accelerated data manipulation and machine learning skills to help address an emergency scenario of national scale.

Optional: Restart the Kernel

If you plan to continue work in other notebooks, please clear GPU memory:

In [ ]:
```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```