Parallel Computing with Dask and Dash

Consider adapting your Dash app to use **Dask** if datasets for your Dash app are too large to fit in memory, or if the performance of your Dash app is limited by the host device's compute power.

Dask is a flexible library for parallel computing in Python which was developed to scale the standard PyData stack (e.g. Numpy, pandas, scikit-learn) and the surrounding ecosystem. It works with the existing Python ecosystem to scale it to multi-core machines and distributed clusters.

Here we provide an overview to help you adapt your Dash app for use with Dask, whether on a cluster or on a single machine.

If you are new to Dask, you may benefit from browsing through **this dask.dataframe tutorial**, as Dash apps with Dask will commonly use `dask.dataframe` in addition to pandas. We also recommend reading the Dask **best practices guide**.

The Dask **documentation** is another good source of information.

Migrating a Dash App to Dask from PyData

Key Differences

Although it is relatively simple to migrate a Dash app to Dask from PyData, it is helpful to understand the underlying concepts behind Dask so that the app can be structured appropriately and run efficiently.

One key difference is that Dask uses what is called "**lazy**" execution, where the evaluation is delayed until explicitly requested. This is carried out mainly by calling the `.compute()` method or the `.persist()` method, which in turn produces an in-memory object such as a number, a numpy array or a pandas dataframe.

For example, a Dask dataframe `groupby` operation is only performed when `df.groupby().compute()` is called. Similarly a Dask array operation to evaluate a mean of `x` would be only performed when `x.mean().compute()` is called.

So you must refactor the Dash app so it can tell Dask *when* to perform its evaluations, prior to updating an appropriate figure, text, or another dashboard component.

Performing the `.compute()` method returns a local in-memory object (such as a pandas dataframe or series object). On the other hand, the `.persist()` method commits the result to the *distributed* memory of the *cluster (**read more**)* for faster loading. (When using a single computer, the `.persist()` method should therefore not be used.)

Additionally, Dask generally deals with large datasets or complex tasks over a cluster, and its tasks are generally computationally expensive. As a result it is even more important to not repeat Dask computations where possible. For instance, any operations that can be performed with an in-memory object should be done so.

We provide some example code for running a Dash app with Dask below, whether on a single computer or on a cluster.

Setting up Dask

Dask can be set up on a single machine without any setup, but running Dask on a cluster requires additional work up front to set up the cluster. Read the **official Dask documentation** for more information.

Install Dask with:

```
pip install dask
pip install pandas  # Required for dealing with dask.dataframe objects
```

You can also install individual subsets of functionalities, such as:

```
pip install "dask[array]"       # Install requirements for dask array
pip install "dask[dataframe]"   # Install requirements for dask dataframe
pip install "dask[distributed]" # Install requirements for distributed dask
```

Dash + Dask on a Single Computer

Migrating to Dask with a single computer configuration such as a laptop is relatively simple. The main changes are to call the `.compute()` method where appropriate.

A very simple Dask/Dash app is shown below, based on the familiar example from the **Layout** chapter.

```
# Run this app with `python app.py` and
# visit http://127.0.0.1:8050/ in your web browser.


from dash import Dash, dcc, html
import plotly.express as px
import dask.dataframe as dd

app = Dash()

df = dd.read_csv('https://gist.githubusercontent.com/chriddyp/5d1ea79569ed194d432e56108a04d188/


def filter_df(pop_thresh):
    filt_df = df[df["population"] > pop_thresh].compute()  # Note the use of .compute() functio
    return filt_df


def build_graphs():
    pop_thresh = 5 * 10 ** 6
    filt_df = filter_df(pop_thresh)

    fig_out = px.scatter(filt_df, x="gdp per capita", y="life expectancy",
                         size="population", color="continent", hover_name="country",
                         log_x=True, size_max=60)
    return fig_out


fig = build_graphs()

app.layout = html.Div([
    dcc.Graph(
        id='life-exp-vs-gdp',
        figure=fig
    )
])

if __name__ == '__main__':
    app.run(debug=True)
```

Here, we filter the original dataframe `df` to generate `filt_df`. Although the original dataframe `df` was a Dask dataframe, the resulting object `filt_df` is a pandas dataframe, as the .compute() method converts the output to an in-memory Pandas dataframe. This app structure allows us to reuse the more compact, in-memory object `filt_df` to generate the figure `fig`. The same object could also be reused if required to generate multiple Plotly figures or any other callback outputs.

In this way, the Dash app can leverage the benefit of Dask for manipulating the Dask dataframe (`df`) while minimizing computationally expensive repetition.

Dash + Dask on a Cluster

You can configure a Dash app (front end) to connect to a Dask cluster (back end), which can be the same device or a separate device. This arrangement separates scalability needs at the front end with the scalability needs of the back end, leading to more efficient use of resources.

There are a number of ways you can run Dask on a cluster including:

- **Coiled**
- **Dask Kubernetes**
- **Dask Cloud Provider**

See the **Dask documentation** for more details.

To use Dash with a Dask cluster, the dataset must be loaded by the cluster and *published* for reuse. Publishing datasets makes the results available for access (**read more**). The Dash app connects to the `scheduler` running on the Dask cluster to perform computations, then retrieves the results to be processed at the front end.

As a more concrete example, the cluster (i.e. the Dask back end) would run:

```
import dask.dataframe as dd
from distributed import Client

scheduler_url = ip_of_scheduler  # Specify IP/URL of the scheduler/cluster (e.g. "127.0.0.1:87
client = Client(scheduler_url)
df = dd.read_csv(path_to_csv).persist()  # Load the dataset into Dask and persist into memory
client.publish_dataset(main_df=df)  # main_df is the name of the published dataset, df the tar
```

Now you can configure the Dash app (front end) to access the cluster with a code pattern similar to:

```
from distributed import Client

# Global initialization
client = None

def init_client():
    """
    This function must be called before any of the functions that require a client.
    """
    global client
    client = Client(scheduler_url)

client = init_client()

df = client.get_dataset(main_df)  # Read the published dataset, calling it by the `main_df` nam
```

In cases where the app uses Dask with a distributed cluster, `df = df.persist()` should be called **prior to publishing** as shown above when the data has been reduced sufficiently to fit into the *distributed* memory. This will make the data available in memory of the cluster for faster loading.

As the front and back ends have been separated, ensure that your app architecture takes the bandwidth between them into consideration. For instance, unnecessarily requesting a large results set from the Dask cluster may degrade performance due to network bottleneck.

Additional Notes

Optimize Distributed Data

Dask dataframes consist of multiple Pandas dataframes (**read more**), also referred to as *partitions*. The amount and arrangement of partitions can significantly impact your app's performance. If your app is slow, you might have too many partitions.

Dask provides a `.repartition()` method so that the dataset may be conveniently repartitioned to suit the cluster. Additionally, the index can be changed with the `.set_index()` method to suit the direction of partitioning.

Review the **best practices guide** for more information on repartitioning.

Categorical Data

As a Dask dataframe consists of multiple pandas dataframes, its behaviors around categorical data depends on whether the dask dataframe is aware of the entire set of possible categories.

Some file types such as parquet files will not retain this metadata. To remedy this problem, you can set up the Dash app to reanalyze categories as appropriate upon loading date such as shown below.

```
df = df.assign(
    cat=df["cat"].astype("category").cat.as_known()
)  # Ensure the categories in this series are known
```

Dtypes

As Dask often deals with large datasets, choosing the right data type may be especially important. We recommend that you review the dataset to determine the right level of precision for each column and assign the dtypes accordingly (e.g. `float32` vs `float64`), such as shown below:

```
df = df.assign(COLUMN=df["COLUMN"].astype("float32"))
```

Dask with GeoDataFrames

Geopandas, and its resulting GeoDataFrames, are often used by many who deal with geospatial data. As geopandas currently does not support use with Dask, consider using **spatialpandas** for parallelized operations with geospatial data.

For the most part, using spatialpandas with Dash is very similar to using dask.dataframe with Dash. Spatialpandas simply allows columns of geometric objects such as points, lines, or polygons to be manipulated with distributed computing as Dask does.

Packing and Performance

Spatialpandas manages multiple partitions of data across a cluster just as is the case with Dask. For similar reasons, efficient packing of these partitions can impact spatialpandas' performance.

For best results, organize partitions based on their spatial proximity using the spatialpandas `DaskGeoDataFrame.pack_partitions` method.