**4x Faster Pandas Operations with Minimal Code Change**

**Stop waiting on pandas operations. Parallelize them.**

One of the major limitations of Pandas is that it can be slow when working with large datasets, particularly when running complex operations. This can frustrate data scientists and analysts who need to process and analyze large datasets in their work.

There are a few ways to address this issue. One way is using parallel processing.



Running multiple pandas operations together! Image by Stable Diffusion.

**Enter Pandarallel**

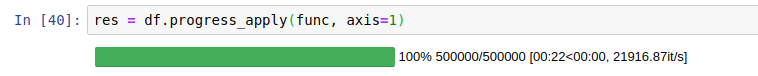
Pandarallel is an open-source Python library that enables parallel execution of Pandas operations using multiple CPUs, resulting in significant speed-ups.

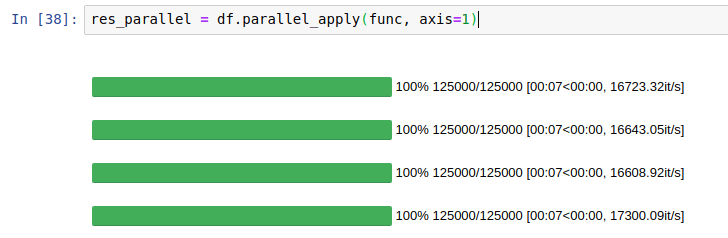
It is built on top of the popular Pandas library, and requires only few code changes to be used.

Disclaimer: I am not affiliated with pandarallel.

**Achieving significant speed-ups with Pandarallel**

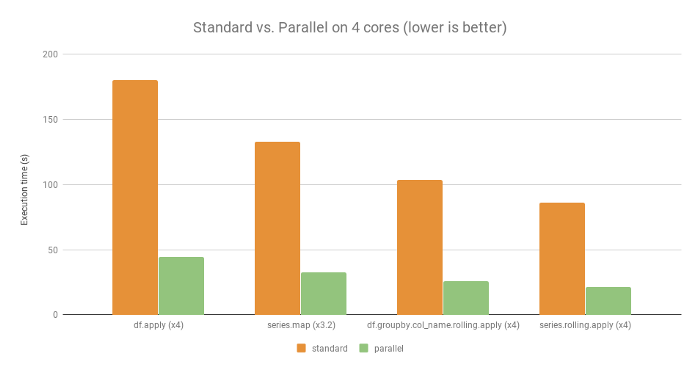
You can see for yourself how much quicker a pandas apply compare with pandarallel’s parallel\_apply. Here, pandarallel distributed the workload across 4 cores.





Source: [Pandarallel’s documentation](https://nalepae.github.io/pandarallel/)

More concretely, the speedup is apparent from the chart below.



4x the speedup. Source: [Pandarallel’s documentation](https://nalepae.github.io/pandarallel/)

**Getting started with Pandarallel**

To install Pandarallel, you can use the pip package manager:

pip install pandarallel

You can import the package into your Python code and initialize it.

from pandarallel import pandarallel  
  
# Initialize pandarallel  
pandarallel.initialize()

Once that is done, you can use the following functions provided.



**Code example of how to use pandarallel**

Let’s first create a mock dataframe that comprises purchases from an e-commerce site. Each row corresponds to the purchase of one product at each date. The columns are:

* date
* product\_id
* quantity

import pandas as pd  
import numpy as np  
  
# Generate a dataframe.  
df = pd.DataFrame()  
  
# Generate a column of random dates from 2019-01-01 to 2019-12-31  
df['date'] = pd.date\_range('2019-01-01', '2019-12-31', periods=10000)  
  
# Seed numpy random   
np.random.seed(0)  
  
# Generate a column of random product\_id from 1 to 5  
df['product\_id'] = np.random.randint(1, 5, 10000)  
  
# Generate a column of quantity bought from 1 to 100  
df['quantity'] = np.random.randint(1, 100, 10000)

Here are the first five rows.

| | date | product\_id | quantity |  
|---:|:--------------------|-----------:|---------:|  
| 0 | 2019-01-01 00:00:00 | 1 | 10 |  
| 1 | 2019-01-01 00:52:25 | 4 | 36 |  
| 2 | 2019-01-01 01:44:50 | 2 | 79 |  
| 3 | 2019-01-01 02:37:15 | 1 | 33 |  
| 4 | 2019-01-01 03:29:41 | 4 | 59 |

Next, let’s use pandarallel to speed up our workflow. To do so, let’s initialize our pandarallel.

from pandarallel import pandarallel  
  
# Initialize pandarallel  
pandarallel.initialize()

Note that you have several options for initializing pandarallel.

# Initialize pandarallel with a progress bar  
pandarallel.initialize(progress\_bar = True)  
  
# Set the number of workers for parallelization.   
# By default, this is the number of cores available.  
pandarallel.initialize(nb\_workers = 4)   
  
# Initialize pandarallel with all logs printed.  
# By default, this is 2 (display all logs), while 0 display n ologs.  
pandarallel.initialize(verbose = 2)

**Use parallel\_apply for applying a function on a column.**

Let’s extract the month from the datecolumn. For example, January is 1, and February is 2. To do so, we can use the parallel\_apply function.

# Group date by month using parallel\_apply  
df['month'] = df['date'].parallel\_apply(lambda x: x.month)

| | date | product\_id | quantity | month |  
|---:|:--------------------|-----------:|---------:|------:|  
| 0 | 2019-01-01 00:00:00 | 1 | 10 | 1 |  
| 1 | 2019-01-01 00:52:25 | 4 | 36 | 1 |  
| 2 | 2019-01-01 01:44:50 | 2 | 79 | 1 |  
| 3 | 2019-01-01 02:37:15 | 1 | 33 | 1 |  
| 4 | 2019-01-01 03:29:41 | 4 | 59 | 1 |

We can also use a lambda function in parallel\_apply. Let’s assign a price for each product\_id. Then, we can calculate the revenue for each transaction.

# Assign a price to each product\_id   
df['price'] = df['product\_id'].parallel\_apply(lambda x: 5.59 if x == 1 else 1.29 if x == 2 else 3.49 if x == 3 else 6.99)  
  
# Get the revenue  
df['revenue'] = df.parallel\_apply(lambda x: x['quantity']\* x['price'], axis=1)

| | date |product\_id |quantity |month |price |revenue |  
|--:|:--------------------|----------:|--------:|-----:|-----:|-------:|  
| 0 | 2019-01-01 00:00:00 | 1 | 10 | 1 | 5.59 | 55.9 |  
| 1 | 2019-01-01 00:52:25 | 4 | 36 | 1 | 6.99 | 251.64 |  
| 2 | 2019-01-01 01:44:50 | 2 | 79 | 1 | 1.29 | 101.91 |  
| 3 | 2019-01-01 02:37:15 | 1 | 33 | 1 | 5.59 | 184.47 |  
| 4 | 2019-01-01 03:29:41 | 4 | 59 | 1 | 6.99 | 412.41 |

**Use parallel\_apply for applying a function on a group.**

You can also group by a particular column before applying parallel\_apply . In the example below, we can group by a particular month, and get the sum of revenue for each month.

# Get the sum of revenue for every month  
monthly\_revenue\_df = df.groupby('month').parallel\_apply(np.sum)[['revenue']]

Better yet, we can perform a rolling sum using parallel\_apply too.

# Generate 3-month rolling revenue by month  
monthly\_revenue\_df['rolling\_3\_mth\_rev'] = monthly\_revenue\_df['revenue'].rolling(3, min\_periods=3).parallel\_apply(np.sum)

| month | revenue | rolling\_3\_mth\_rev |  
|--------:|----------:|--------------------:|  
| 1 | 188268 | nan |  
| 2 | 164251 | nan |  
| 3 | 176198 | 528717 |  
| 4 | 178021 | 518470 |  
| 5 | 188940 | 543159 |

**Use parallel\_applymap for applying a function to the entire dataframe**

If there is a function to apply to the entire dataframe, applymap is the ideal function. For example, to convert all the elements of df to a string, we can use this function.

# Convert every element of df to a string  
df.parallel\_applymap(lambda x: str(x))

**The complete code**

import pandas as pd  
import numpy as np  
from pandarallel import pandarallel  
  
# Generate a dataframe.  
df = pd.DataFrame()  
  
# Generate a column of random dates from 2019-01-01 to 2019-12-31  
df['date'] = pd.date\_range('2019-01-01', '2019-12-31', periods=10000)  
  
# Seed numpy random   
np.random.seed(0)  
  
# Generate a column of random product\_id from 1 to 5  
df['product\_id'] = np.random.randint(1, 5, 10000)  
  
# Generate a column of quantity bought from 1 to 100  
df['quantity'] = np.random.randint(1, 100, 10000)  
  
# Initialize pandarallel  
pandarallel.initialize()  
  
# Group date by month using parallel\_apply  
df['month'] = df['date'].parallel\_apply(lambda x: x.month)  
  
# Assign a price to each product\_id   
df['price'] = df['product\_id'].parallel\_apply(lambda x: 5.59 if x == 1 else 1.29 if x == 2 else 3.49 if x == 3 else 6.99)  
  
# Get the revenue  
df['revenue'] = df.parallel\_apply(lambda x: x['quantity']\* x['price'], axis=1)  
  
# print(df.head().to\_markdown())  
  
# Get the sum of revenue for every month  
monthly\_revenue\_df = df.groupby('month').parallel\_apply(np.sum)[['revenue']]  
  
# Generate 3-month rolling revenue by month  
monthly\_revenue\_df['rolling\_3\_mth\_rev'] = monthly\_revenue\_df['revenue'].rolling(3, min\_periods=3).parallel\_apply(np.sum)  
  
# print(monthly\_revenue\_df.head().to\_markdown())

**When NOT to use pandarallel**

We should not use Pandarallel when the data cannot be fit into memory. In that case, usesparkorpyspark or vaex .

Read my article on using [**Vaex to process 9 billion rows per second**](https://medium.com/p/a272d3485b24).

That said, there are multiple use cases of pandarallel that would benefit data scientists. Quit waiting on your pandas operations and parallelize them.