Project Elective Report

AI for Big Geo-Spatial Data :

Scaling things out

By
Pratik Ratnadeep Ahirrao (IMT2019064)
Pratyush Upadhyay (IMT2019066)

1 Introduction

1.1 Problem Statement

The task is to reduce the computation time of calculating MNDWI and NDVI indices for satellite images from the *DynamicEarthNet* Dataset and storing them in GeoTIFF format using Dask. *DynamicEarthNet* is the first dataset that provides this unique combination of daily measurements and high-quality labels. It consists of daily, multi-spectral satellite observations of 75 selected areas of interest (AOIs) distributed over the globe with imagery from Planet Labs. These observations are paired with pixel-wise monthly semantic segmentation labels of 7 land use and land cover (LULC) classes. The dataset can be downloaded from DynamicEarthNet. Out of these 75 AOIs, we will focus mainly on two AOIs i.e. *planet16N* and *planet18N*.

1.2 Installation Steps

The jupyter notebook can be downloaded from the Github repository. Before running the notebook, create two folders *ndvi-images* and *mndwi-images* in the same directory. The following modules/libraries should be installed in your system before running the notebook:

rasterio: A Python module for reading and writing geospatial raster data. It allows users to work with a wide variety of raster data formats, including GeoTIFF.

```
# pip and python must be installed in your system
pip install rasterio
```

Dask: Dask is a parallel computing library for Python that enables parallel and distributed computing across multiple CPU cores or clusters. It allows users to work with large datasets that are too big to fit into memory on a single machine, by breaking them into smaller partitions and distributing them across multiple processors or machines.

```
# pip and python must be installed in your system
pip install dask
```

Pytorch: PyTorch is a Python package that helps in Tensor computation (like NumPy) with strong GPU acceleration.

```
# pip and python must be installed in your system
pip install torch
```

2 Calculations & Analysis

We want to calculate ndvi and mndwi indexes of the two AOIs and store them in GeoTIFF formats. Using this we can analyze the changes in the AOI over a given period of time.

Normalized Difference Vegetation Index (NDVI)

The NDVI is a dimensionless index that describes the difference between visible and near-infrared reflectance of vegetation cover and can be used to estimate the density of green on an area of land. NDVI always ranges from -1 to +1. But there isn't a distinct boundary for each type of land cover. For example, when you have negative values, it's highly likely that it's water. On the other hand, if you have an NDVI value close to +1, there's a high possibility that it's dense green leaves. But when NDVI is close to zero, there are likely no green leaves and it could even be an urbanized area.

Normalized Difference Vegetation Index (NDVI) uses the NIR and red channels in its formula.

$$NDVI = \frac{NIR - Red}{NIR + Red}$$

Modified Normalized Difference Water Index (MNDWI)

It is a vegetation index that is used to identify water bodies in remote sensing images. MNDWI is a modification of the NDWI (Normalized Difference Water Index) that takes into account the presence of vegetation.

Like NDVI, MNDWI is calculated using the difference between two spectral bands, in this case, the green and mid-infrared bands. The formula for MNDWI is:

$$MNDWI = \frac{Green - NIR}{Green + NIR}$$

The resulting value ranges from -1 to +1, with values close to -1 indicating the presence of water and values close to +1 indicating the presence of vegetation.

MNDWI indexes are commonly used in remote sensing applications to map and monitor water bodies such as lakes, rivers, and wetlands. It is also used to detect changes in water bodies over time, such as changes in water level, water quality, and extent of water bodies due to natural or human-induced factors.

We can see changes in the AOIs (as we move from left to right) using the ndvi and mndwi indexes over a period of time.
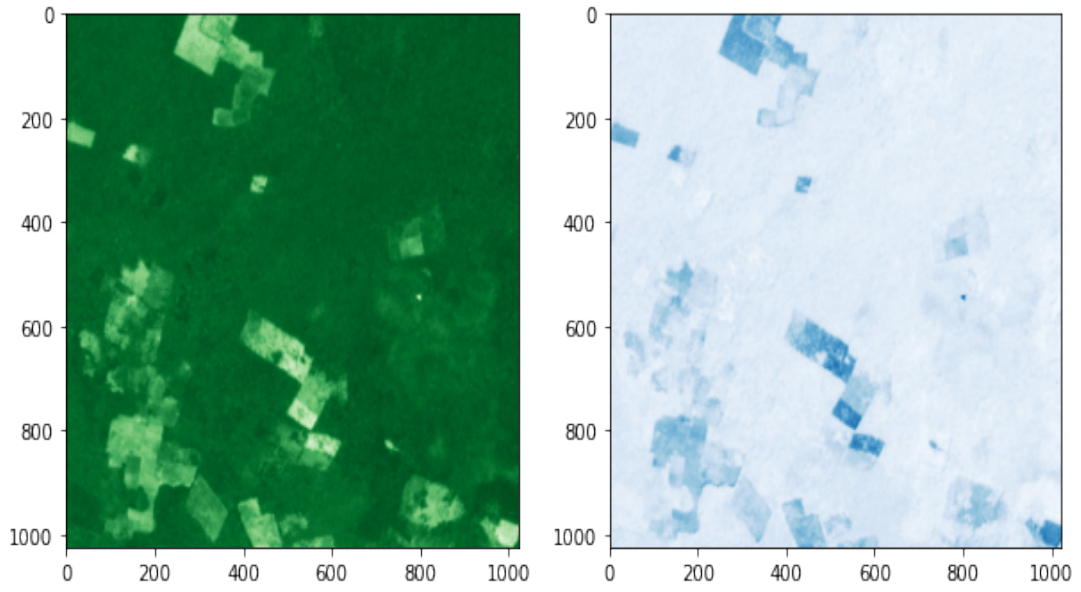
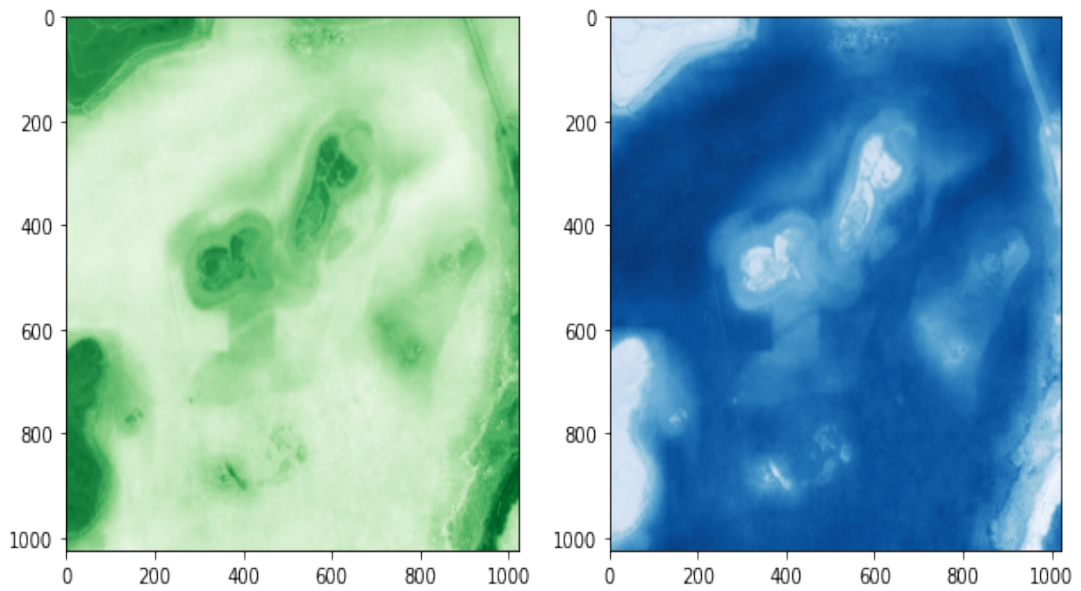Figure 1: NDVI and MNDWI indexes of **16N-2018-01-01.tif** image

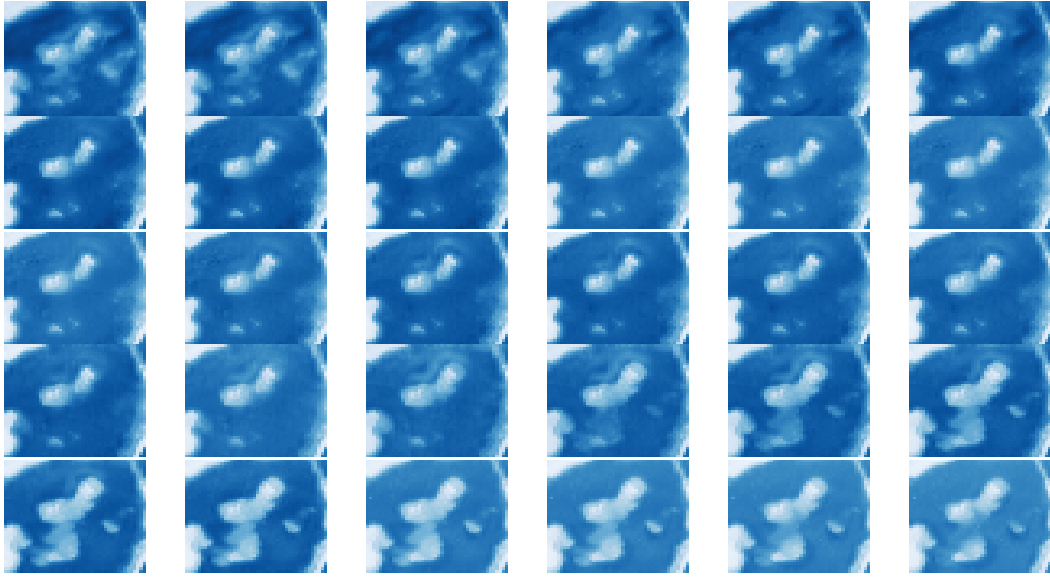Figure 2: NDVI and MNDWI indexes of **18N-2018-01-01.tif** image

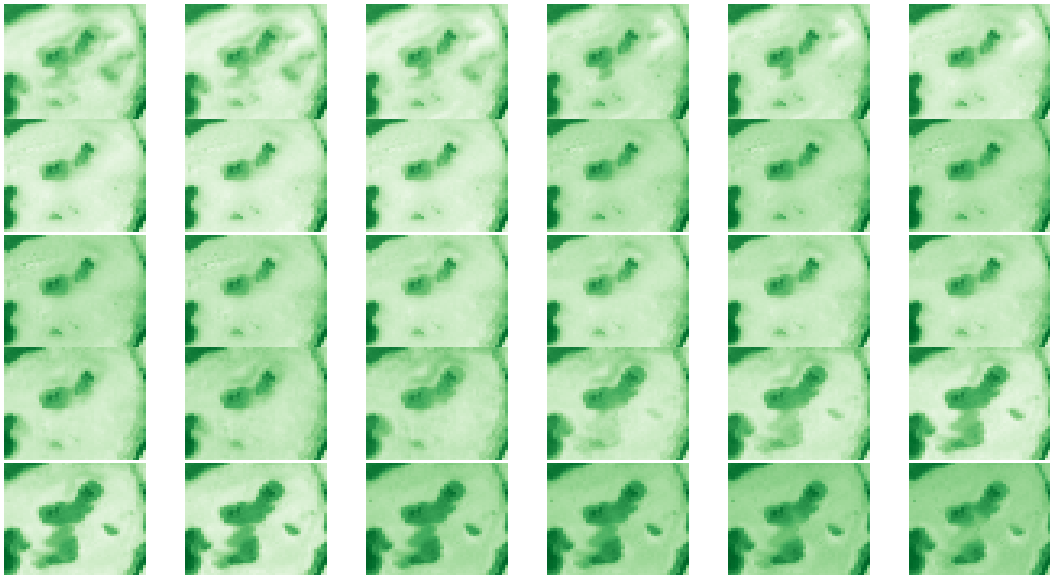Figure 3: MNDWI indexes of **18N** AOI over a month of time

Figure 4: NDVI indexes of **18N** AOI over a month of time

Figure 5: MNDWI indexes of 18N AOI over a period of two years. Here each image is the first day of each month

Figure 6: NDVI indexes of 18N AOI over a period of two years. Here each image is the first day of each month

Figure 7: MNDWI indexes of **16N** AOI over a month of time

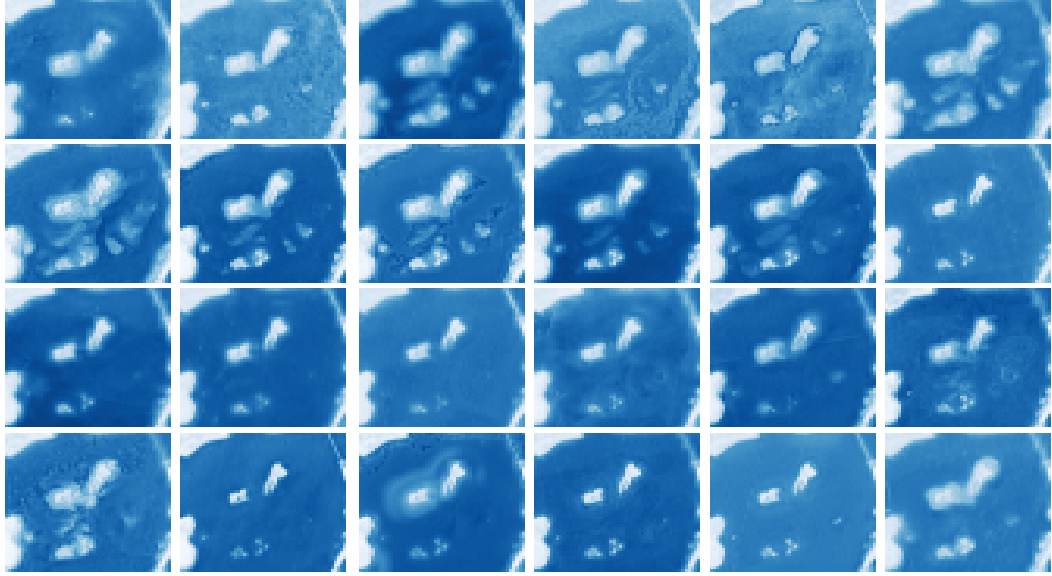Figure 8: NDVI indexes of **16N** AOI over a month of time

Figure 9: MNDWI indexes of 16N AOI over a period of two years. Here each image is the first day of each month
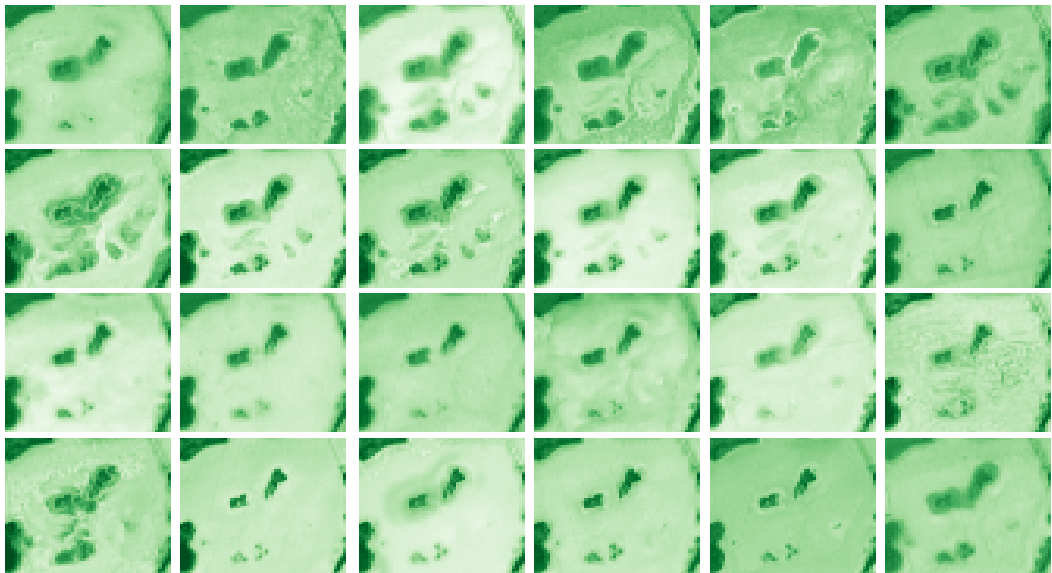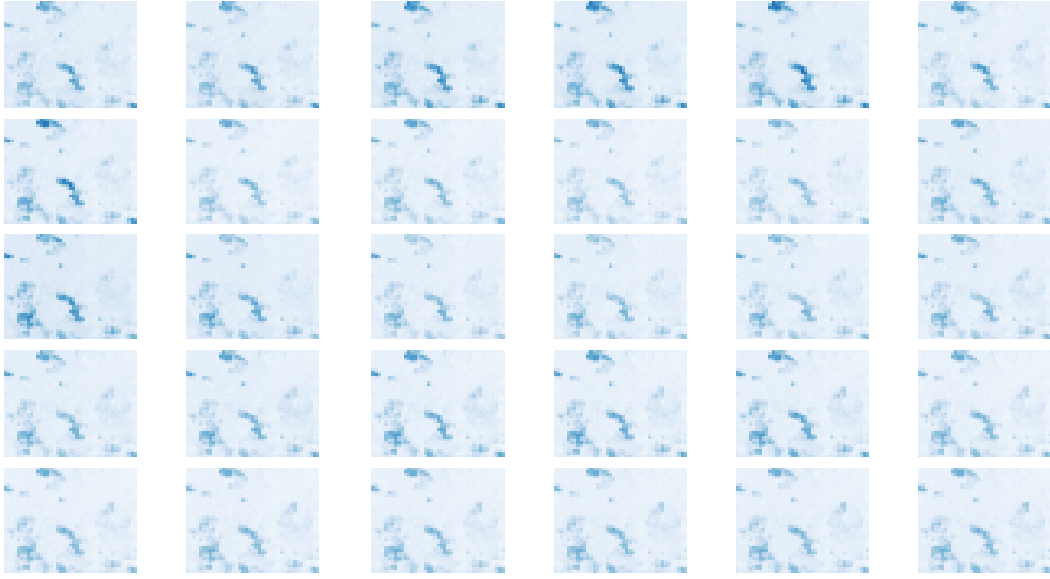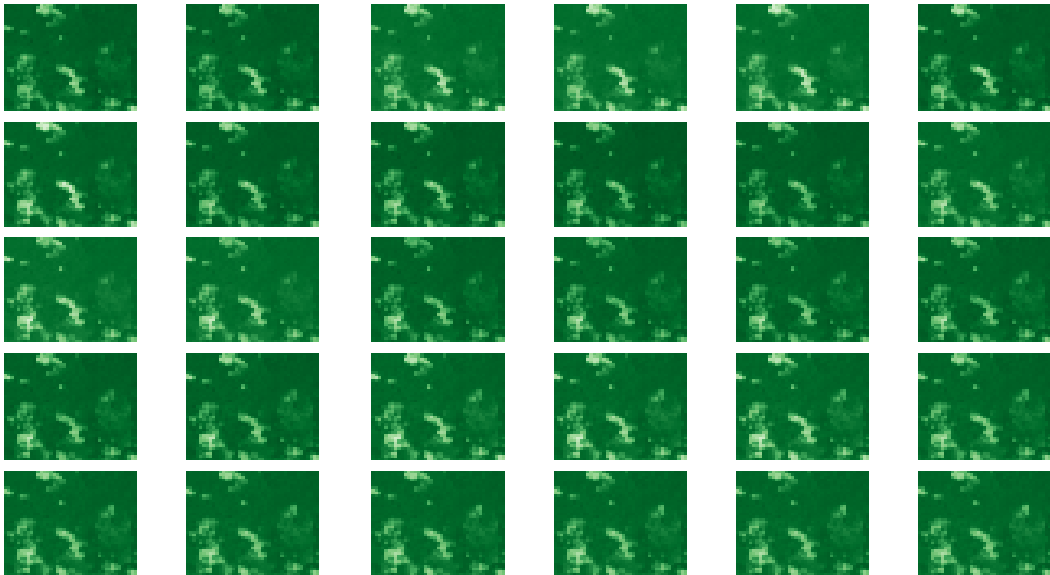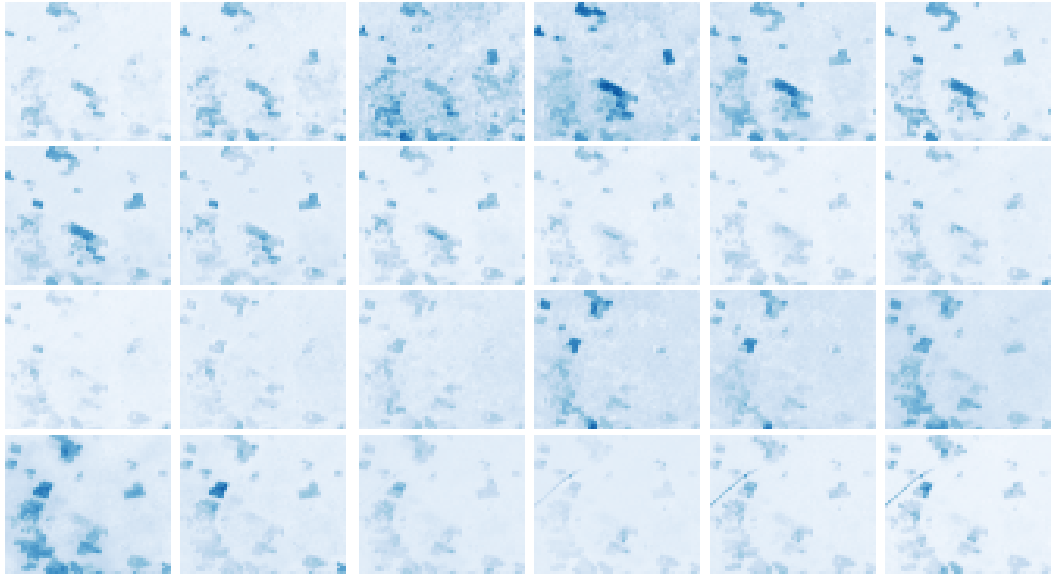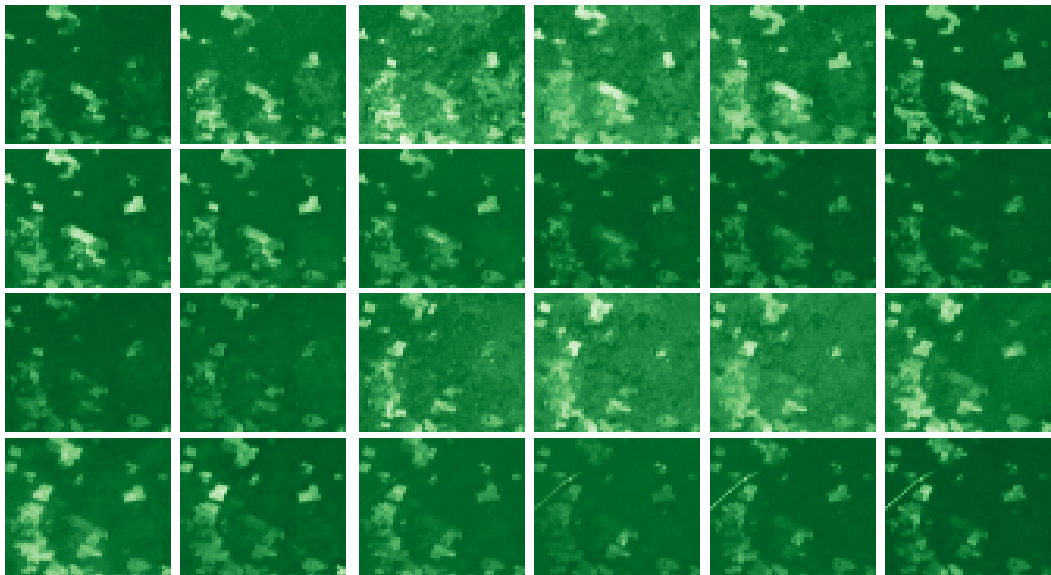

Figure 10: NDVI indexes of 16N AOI over a period of two years. Here each image is the first day of each month

3   Dask

Python packages like numpy, pandas, sklearn, etc. make data manipulation and ML tasks very convenient. For most data analysis tasks, the python pandas package is good enough. You can do all sorts of data manipulation and is compatible with building ML models. But as our data gets bigger and bigger, it can be difficult to fit the data in the RAM. This is where Dask comes in.

Dask is an open-source library that provides advanced parallelization for analytics, especially when you are working with large data. It is built to help you improve code performance and scale up without having to rewrite your entire code. The good thing is, you can use all your favorite Python libraries as Dask is built-in coordination with numpy, pandas, and others.
We will make use of dask arrays instead of numpy arrays. But what can be its advantages?

A dask array resembles a numpy array in its look and feel. A dask array doesn't actually store any data, though. The computations required to produce the data are instead symbolically represented. Nothing is actually computed until the actual numerical values are needed. This mode of operation is called "lazy"; it allows one to build up complex, large calculations symbolically before turning them over to the scheduler for execution.
If we want to create a NumPy array of all ones, we do it like this:

```python
import numpy as np
shape = (1000, 4000)
ones_np = np.ones(shape)
ones_np
```

```python
# output Obtained
array([[1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       ...,
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.],
       [1., 1., 1., ..., 1., 1., 1.]])
```
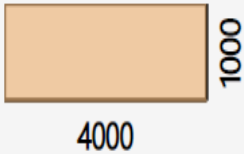
This array contains exactly 32 MB of data:

```python
ones_np.nbytes / 1e6
```

```python
32.0
```

Now let's create the same array using dask's array interface.

```
import dask.array as da
ones = da.ones(shape)
ones
```

	Array	Chunk
Bytes	30.52 MiB	30.52 MiB
Shape	(1000, 4000)	(1000, 4000)
Count	1 Tasks	1 Chunks
Type	float64	numpy.ndarray

4000
1000

It's difficult to say for certain whether the Dask computation time will always be better than that of NumPy, as it depends on a variety of factors such as the size of the input data, the number of cores available, and the specific computation being performed. However, in general, Dask can often be faster than NumPy when dealing with large datasets that do not fit into memory, as Dask can split the computation across multiple threads or processes and operate on the data in chunks.

As we have large dataset, we will initialize a Dask client and using its distributed computing capabilities to speed up the computation. This can allow us to take advantage of multiple cores or machines to perform the computation in parallel, reducing the time required for the task.

Importing client and local cluster:

```
from dask.distributed import Client, LocalCluster
cluster = LocalCluster()
client = Client(cluster)
client
```

This creates a cluster with four worker nodes by default. The worker nodes are used to parallelize the computation across multiple threads or processes. When we create a Dask client using *dask.distributed.Client()*, we are initializing a cluster of worker nodes that can be used to distribute the computation across multiple machines or cores.

The use of worker nodes in Dask is only relevant if you are initializing a Dask client and distributing the computation across multiple threads or processes. If you are not using Dask, or are using it in a single-threaded mode, worker nodes are not applicable.

The generated *Client* object connects to the scheduler node and provides a convenient interface for submitting tasks to the cluster. Once the Dask cluster is set up, we can submit tasks to it using the *dask.compute* function, just as before. Dask will automatically distribute the tasks among the available worker nodes, so the computation will be performed in parallel across multiple cores or threads.

Dask Dashboard

The Dask dashboard is a web-based interface that provides real-time monitoring and diagnostics for Dask computations. It allows users to visualize the status of their Dask computations, including information on task execution, memory usage, and data transfer rates.

To open the Dask dashboard, you need to first start a Dask cluster and connect to it using a Client object.

```python
from dask.distributed import Client

client = Client()  # connect to an existing Dask cluster

# open the dashboard in a new browser tab
client.dashboard_link()
```

CPU Use (%)

Memory Use (%)

name	address	nthreads	cpu	memory	limit	memory %	managed	unmanag	unmanag	spilled	# fds	net read	net write	disk read	disk write
Total (4)		12	0 %	4.7 GiB	19.8 GiB	23.9 %	0.0	3.7 GiB	1.1 GiB	0.0	0	41 KiB	16 KiB	28 MiB	160 KiB
0	tcp://127.	3	0 %	1.2 GiB	4.9 GiB	23.6 %	0.0	929.9 Mi	268.3 Mi	0.0	0	12 KiB	5 KiB	13 MiB	0
1	tcp://127.	3	0 %	1.3 GiB	4.9 GiB	25.3 %	0.0	915.1 Mi	368.0 Mi	0.0	0	22 KiB	5 KiB	15 MiB	0
2	tcp://127.	3	0 %	1.1 GiB	4.9 GiB	22.7 %	0.0	972.7 Mi	177.2 Mi	0.0	0	4 KiB	4 KiB	262 KiB	0
3	tcp://127.	3	0 %	1.2 GiB	4.9 GiB	23.8 %	0.0	919.9 Mi	288.7 Mi	0.0	0	3 KiB	2 KiB	184 KiB	160 KiB

Shut down the cluster

It is important to close the Dask cluster when you are done with your computation or analysis for saving computation state and freeing up resources.

```
from dask.distributed import Client

client = Client()  # connect to existing Dask cluster
# do some computation using the client

client.close()  # close the client and shut down the cluster
```

We have also used *dask.delayed* function to lazily evaluate each iteration of the loop as a separate task. This will allow Dask to schedule the tasks in parallel and execute them on a cluster or on multiple threads or processes on a single machine. Here is an example of using the delayed function:

```
import dask
from dask import delayed

@delayed
def add(a, b):
    return a + b

a = delayed(1)
b = delayed(2)
c = add(a, b)

result = c.compute()
print(result)
```

High-level overview of the working of dask cluster:

- In Dask, data is divided into smaller, manageable chunks called "task graphs". These task graphs are then distributed across multiple cores or machines for parallel processing.

- When you perform an operation on a Dask array or dataframe, Dask automatically breaks the computation down into a series of smaller tasks or subtasks, each of which operates on a chunk of the data. These subtasks are then scheduled and executed in parallel across multiple cores or machines using a scheduler.

- The scheduler decides how to allocate these subtasks to the available workers, taking into account factors such as the availability of resources, the dependencies between tasks, and the overall efficiency of the computation. The scheduler also handles communication and data transfer between the workers, ensuring that data is moved to the appropriate worker when needed and minimizing the amount of data that needs to be transferred.

- Once a worker has completed a subtask, the results are combined and passed back to the scheduler, which then schedules the next set of subtasks based on the updated state of the computation. This process continues until all of the subtasks have been completed and the final result of the computation is returned.
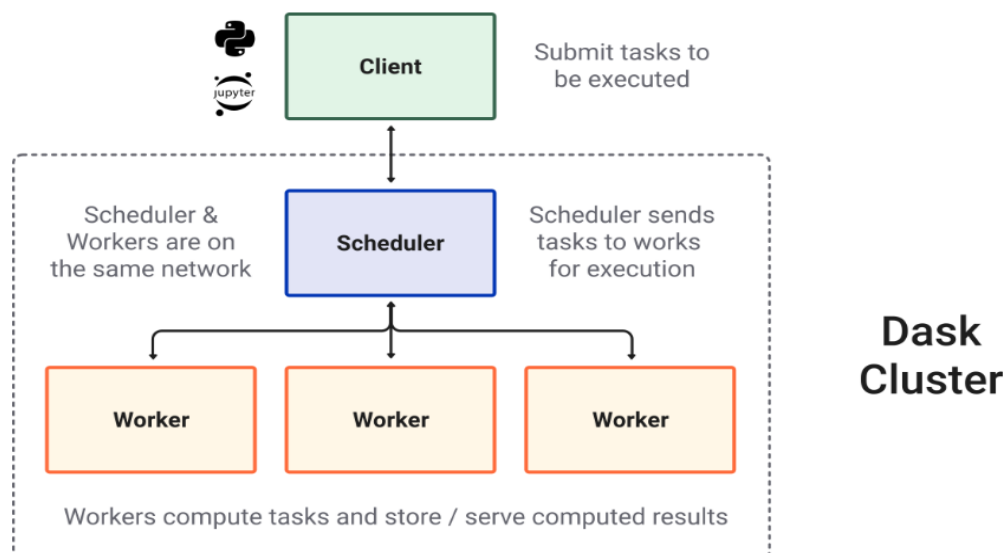
4 Experiments & Results

We create a Dask cluster locally and specify the worker nodes, depending on the complexity of our task we can change it accordingly. The task that we are doing currently is finding the ndvi and mndwi indexes and storing them in the form of *.tif* image for a given set of images belonging to a particular Area of Interest.

We compare the *Computation time* and *Wall time* using dask and without dask for both the AOIs. *Computation time* measures the actual time spent by the CPU on computations, while *Wall time* measures the total time from the start to the end of a program's execution, considering all factors that affect its execution. *Wall time* is typically higher than *Computation time* due to factors such as I/O operations, waiting for resources, and other delays.

We perform the above task without using Dask and then using Dask for different numbers of worker nodes. There might be minor fluctuations in results when re-running for the notebook but the overall comparable difference is significant.

4.1 For 18N AOI

Calculating ndvi indexes without Dask:

Computation Time	Wall Time
1min 22s	2min 8s

Calculating ndvi indexes using Dask:

Worker Nodes	Computation Time	Wall Time
1	1.58 s	1min 23s
2	1.67 s	40.9 s
3	2.06 s	36.7 s
4	4.25 s	33.2 s
5	2.75 s	26.8 s
6	4.59 s	30.7 s
7	6.78 s	29.2 s

Calculating mndwi indexes without Dask:

Computation Time	Wall Time
1min 39s	2min 7s

Calculating mndwi indexes using Dask:

Worker Nodes	Computation Time	Wall Time
1	1.95 s	1min 25s
2	1.67 s	41.2 s
3	1.44 s	34.3 s
4	1.16 s	32.5 s
5	1.92 s	26.3 s
6	4.53 s	30.2 s
7	5.7 s	29.7 s

4.2 For 16N AOI

Calculating ndvi indexes without Dask:

Computation Time	Wall Time
57.2 s	1min 37s

Calculating ndvi indexes using Dask:

Worker Nodes	Computation Time	Wall Time
1	578 ms	1min 3s
2	1.17 s	43.3 s
3	1.03 s	31 s
4	1.91 s	32.1 s
5	4 s	29.6 s
6	2.86 s	26.4 s
7	6.59 s	33.2 s

Calculating mndwi indexes without Dask:

Computation Time	Wall Time
52.5 s	1min 38s

Calculating mndwi indexes using Dask:

Worker Nodes	Computation Time	Wall Time
1	1.39 s	1min 2s
2	1.2 s	48.3 s
3	906 ms	30 s
4	1.67 s	30.5 s
5	4.69 s	32.4 s
6	3 s	30.5 s
7	6.03 s	29.5 s

5   Observation

- We can see the significant difference in computation time(Wall time) for both AOI's with Dask and without Dask.

- For both the AOI dataset we can observe that the overall computation time decreases when we increase our worker nodes up to 4 or 5, after that it remains more or less the same and then slightly increases. This may be caused due to run-time overhead of running many worker nodes, which also further increases their internal communication time over the network.

6   Conclusion

Using Dask we can easily parallelise any computationally complex and heavy task and thus reduce the overall computation time. The more complex our task, the more worker nodes will be needed to parallelize it, and if the task is simple or deals with a small dataset, large clusters might not reduce the overall computation time due to run-time overheads and intra-node communication. So it's a good practice to find the optimal number of worker nodes for any given task that reduces the time significantly.

7 References

1 Dask Tutorials
   https://www.machinelearningplus.com/python/dask-tutorial/

2 Dask Arrays
   https://earth-env-data-science.github.io/lectures/dask/dask_arrays.html

3 Dask Documentation
   https://tutorial.dask.org/00_overview.html

4 Normalized Difference Vegetation Index
   https://gisgeography.com/ndvi-normalized-difference-vegetation-index/