Python for Data Sciences

Parallelizing: Ray, Dask

- Armando Camerlingo

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani

Parallelization

ITINERIS

Numba

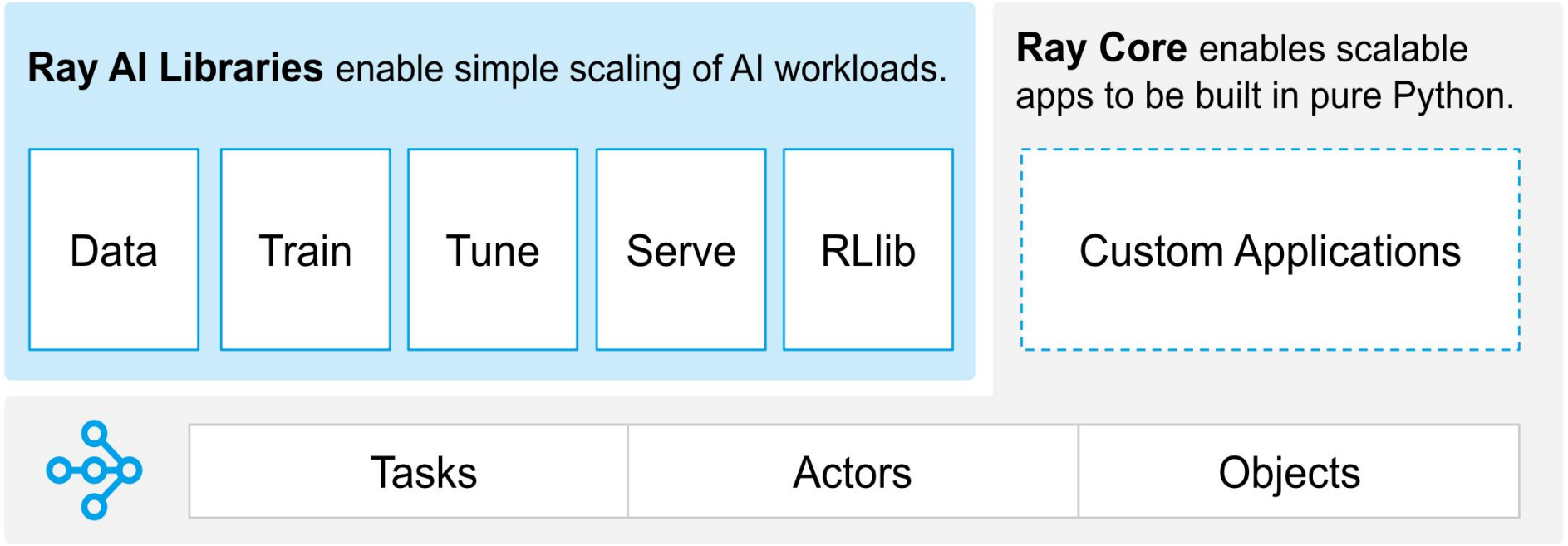
Ray

Dask

RAY

Ray enables arbitrary Python functions to be executed asynchronously on separate Python workers. These asynchronous Ray functions are called “tasks.” The cluster scheduler distribute tasks across the cluster for parallelized execution.

RAY



Data: Scalable Datasets for ML
Train: Distributed Training
Tune: Scalable Hyperparameter Tuning
RLlib: Scalable Reinforcement Learning
Serve: Scalable and Programmable Serving

Tasks: Stateless functions executed in the cluster.
Actors: Stateful worker processes created in the cluster.
Objects: Immutable values accessible across the cluster.

RAY

Let’s install RAY and load all the packages that we will need

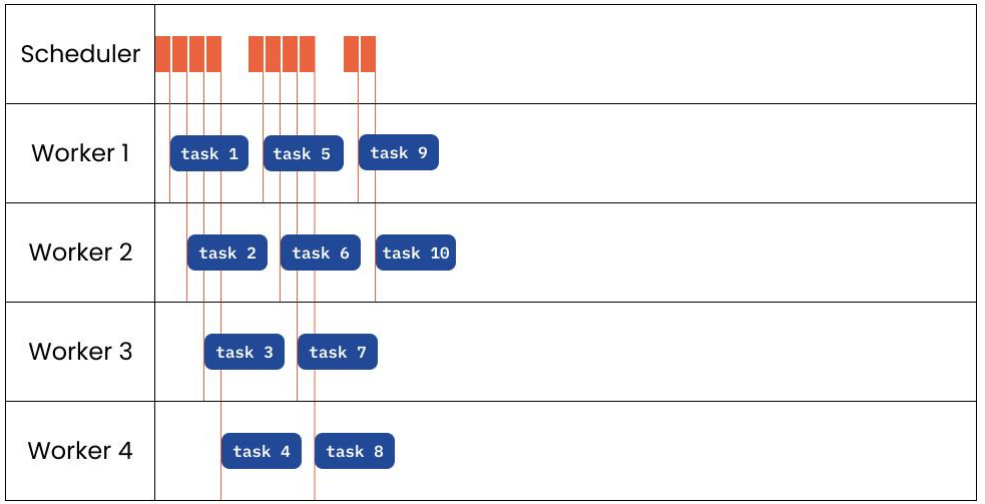
```
!uv pip install ray
```

```
import os
import time
import logging
import math
import random
from pathlib import Path
from typing import Tuple, List
import numpy as np
import ray
```

RAY: serial vs parallel execution

Serial tasks as regular Python functions are executed in a sequential manner, as shown in the diagram below. If I launch ten tasks, they will run on a single worker, one after the other.

Compared to serial execution, a Ray task executes in parallel, scheduled on different workers. The Raylet will schedule these task based on scheduling policies.



RAY

RAY use the decorator `@ray.remote` to generate stateless tasks, that will be schedulated on a RAY cluster node. All the scheduling is taken care by RAY, we don’t need to specify anything.

When using the decorator, we have to modify a little our scripts:

- Invocation: The regular version is called with func_name(), whereas the remote Ray version is called with func_name.remote().

- Mode of execution and return values: A Python func_name() executes synchronously and returns the result of the function, whereas a Ray task func_name.remote() immediately returns an ObjectRef (a future) and then executes the task in the background on a remote worker process. The result of the future is obtained by calling ray.get(ObjectRef).

RAY: launch the cluster

We can check how many cores are in the machine on which our notebook is running:

```
import multiprocessing

cores = multiprocessing.cpu_count() # Count the number of cores in a
computer
cores
```

```
2
```

Unfortunately, the free version of Google Colab provides only 2 cores notebooks and we also won't be able to see the dashboard. But we'll see how to prepare our code to run on a RAY cluster and how to launch it.

```
context = ray.init(num_cpus=1,logging_level=logging.ERROR)
print(context.dashboard_url)
```

RAY: example 1

Let's estimate the value of pi using a Monte Carlo method, the function we have seen also in previous lesson.
Given we know that the true ratio to be pi/4, we can multiply our estimated ratio by 4 to approximate the value of pi.

```
def sampling_task(num_samples: int, task_id: int) -> int:
    np.random.seed(task_id)  # Ensure different seed for each call
    x = np.random.rand(num_samples)
    y = np.random.rand(num_samples)
    num_inside = np.sum(x**2 + y**2 <= 1.0)

    return num_inside, task_id
```

RAY: example 1

Set up the values for our run

```
NUM_SAMPLING_TASKS = 2 #NUM_OF_CORES
NUM_SAMPLES_PER_TASK = 1_000_000
TOTAL_NUM_SAMPLES = NUM_SAMPLING_TASKS * NUM_SAMPLES_PER_TASK
```

First we create a function that computes pi from all the results produced bu the various tasks:

```
def calculate_pi(results):
    total_num_inside = 0
    for task_id, num_inside in results:
        print(f"Task id: {task_id} | Samples in the circle: {num_inside}")
    total_num_inside = sum(num_inside for num_inside, _ in results)
    pi = (total_num_inside * 4) / TOTAL_NUM_SAMPLES
    return pi
```

RAY: example 1

Define a function that performs tasks in serial

```
def run_serial(sample_size) -> List[int]:
    results = [sampling_task(sample_size, i+1) for i in
range(NUM_SAMPLING_TASKS)]
    return results
```

Now let’s decorate oour sampling function for ray and define a function that performs the tasks in parallel

```
@ray.remote
def sample_task_distribute(sample_size, i) -> object:
    return sampling_task(sample_size, i)
```


```
@ray.remote
def sample_task_distribute(sample_size, i) -> object:
    return sampling_task(sample_size, i)
```

RAY: example 1

Serial run

```
print(f"Running {NUM_SAMPLING_TASKS} tasks serially....")
```

```
Running 16 tasks serially....
```

```
%%time
results = run_serial(NUM_SAMPLES_PER_TASK)
pi = calculate_pi(results)
print(f"Approximation for π: {pi:5f}")
```

```
Task id: 785900 | Samples in the circle: 1
Task id: 784951 | Samples in the circle: 2
Approximation for π: 3.141702
CPU times: user 55.1 ms, sys: 26.8 ms, total: 81.9 ms
Wall time: 84.7 ms
```

RAY: example 1

Parallel run with RAY distributed tasks

```
%%time
results = run_disributed(NUM_SAMPLES_PER_TASK)
pi = calculate_pi(results)
print(f"Estimated value of π is: {pi:5f}")
```

```
Task id: 785900 | Samples in the circle: 1
Task id: 784951 | Samples in the circle: 2
Estimated value of π is: 3.141702
CPU times: user 4.98 ms, sys: 420 µs, total: 5.4 ms
Wall time: 68 ms
```

Faster! baseline = 55 ms

RAY: example 1 (alternative)

We can also use math functions instead of numpy:

```
def sampling_task(num_samples: int, task_id: int) -> int:
    num_inside = 0
    for i in range(num_samples):
        x, y = random.uniform(-1, 1), random.uniform(-1, 1)
        # check if the point is inside the circle
        if math.hypot(x, y) <= 1:
            num_inside += 1
    return num_inside, task_id
```

```
%%time
results = run_disributed(NUM_SAMPLES_PER_TASK)
pi = calculate_pi(results)
print(f"Estimated value of π is: {pi:5f}")
```

```
CPU times: user 8.53 ms, sys: 44 µs, total: 8.58 ms
Wall time: 1.22 s
```

Slower than Numpy...
Why?

Exercise

We used NUM_SAMPLES_PER_TASK = 1_000_000. What happens to the value of pi and the run times if we use 100_000 samples?

Exercise

We used NUM_SAMPLES_PER_TASK = 1_000_000. What happens to the value of pi and the run times if we use 100_000 samples?

Solution

```
Approximation for π: 3.145220
CPU times: user 6.9 ms, sys: 3.03 ms, total: 9.93 ms
Wall time: 9.51 ms
```

```
Estimated value of π is: 3.143480
CPU times: user 2.55 ms, sys: 2.51 ms, total: 5.06 ms
Wall time: 128 ms
```

- Worse pi approximation;
- Less performance gain from running it in distributed mode;

Dask

Dask is a Python library for parallel and distributed computing. Dask is:

- Easy to use and set up (it’s just a Python library);
- Powerful at providing scale, and unlocking complex algorithms;
- It parallelizes for loops;

Dask

For now, let’s see the default implementation. We’ll use two simple function to play with Dask:

```
from time import sleep

def inc(x):
    sleep(1)
    return x + 1

def add(x, y):
    sleep(1)
    return x + y
```

Dask: sequential run

We time a run of both functions:

```
%%time

x = inc(1)
y = inc(2)
z = add(x, y)
```

```
CPU times: user 9.54 ms, sys: 2.01 ms, total: 11.6 ms
Wall time: 3 s
```

Obviously, it took 3 seconds because we run them sequentially. But there’s nothing keeping us to run the two inc function calls in parallel.

Dask: Delayed

We can use the `dask.delayed` function to transform the two functions. These new functions don't compute as soon as we call them , but it keeps track of the function and its arguments. So when we run the following code, it is (almost) immediate.

```
from dask import delayed
import dask
```

```
%%time
x = delayed(inc)(1)
y = delayed(inc)(2)
z = delayed(add)(x, y)
```

```
CPU times: user 9.63 ms, sys: 1.37 ms, total: 11 ms
Wall time: 2 s
5
```

It runs faster than 3 sec!

Dask: Delayed

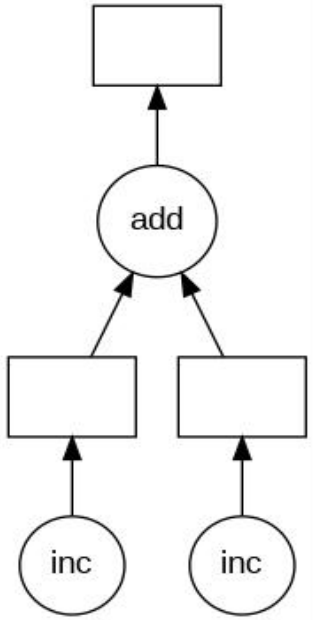
z is a delayed object:

```
z
```

```
Delayed('add-86892dee-41ff-4aad-a9cb-7e9fcd473863')
```

There is also a method that produces a graph of this object:

```
z.visualize()
```



Dask: Delayed

A few questions arise:

- Why did we go from 3s to 2s? Why not 1s?
- What would happen if no sleep() was present in the function?
- What if we want to compute multiple outputs?

Dask Array

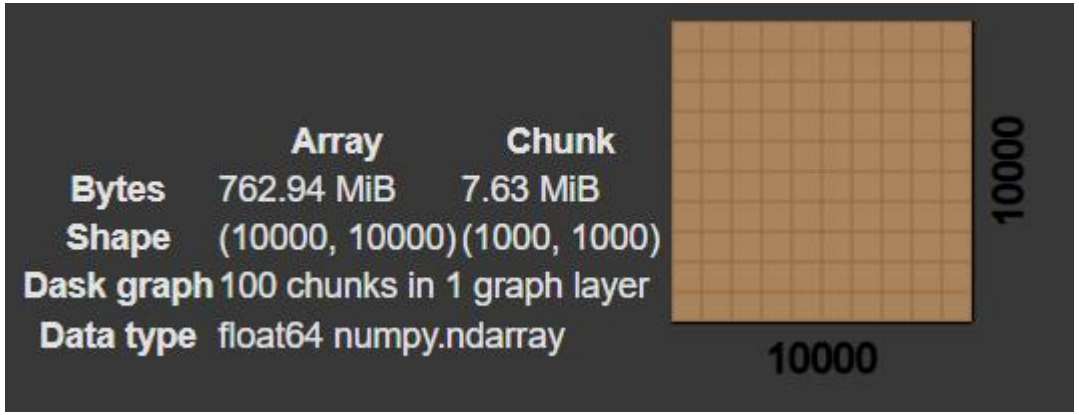
Dask has its own array implementation:

- Parallel: tries to use all the cores avaiable;
- Let you manage data larger than your available RAM memory by partitioning it in smaller arrays;
- Algorithms by block: complex computations divided in smaller ones;

Dask Array

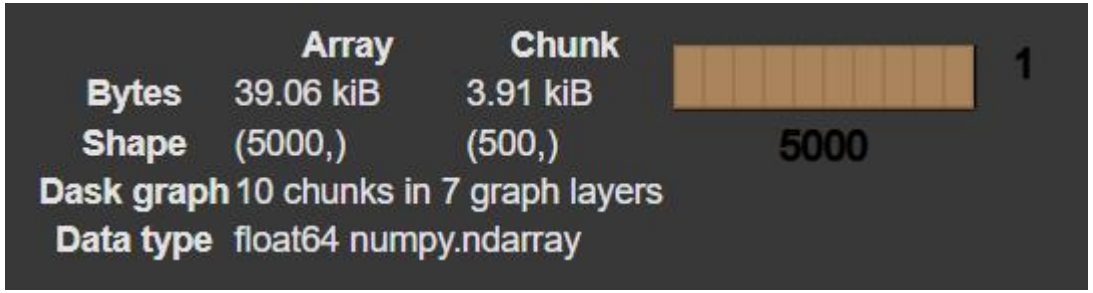
```
import dask.array as da
import numpy as np
import dask
```


```
import dask.array as da
import numpy as np
import dask
```

Dask Array

```
y = x + x.T
z = y[::2, 5000:].mean(axis=1)
z
```




```
z.compute()
```

```
array([0.99362216, 1.01060217, 1.01173196, ..., 1.00714347, 1.00102091,
       0.99252842])
```

Dask Array vs Numpy Array

```
%%time
xn = np.random.normal(10, 0.1, size=(30_000, 30_000))
yn = xn.mean(axis=0)
yn
```

```
CPU times: user 30.7 s, sys: 1.5 s, total: 32.2 s
Wall time: 34.2 s
array([10.00016485, 10.00000635, 10.00003001, ...,  9.99937723,
        9.99936001, 10.00029068])
```

```
%%time
xd = da.random.normal(10, 0.1, size=(30_000, 30_000), chunks=(300, 300))
yd = xd.mean(axis=0)
yd.compute()
```

```
CPU times: user 1min 7s, sys: 1.11 s, total: 1min 9s
Wall time: 42 s
array([10.00017311,  9.99991263,  9.99973077, ...,  9.99933366,
        9.99966248,  9.99997549])
```

Dask Dataframes

As in Pandas, dataframes are implemented in Dask and leverage the same infrastructure that run dask.delayed.

- Pandas dataframes saves the entire dataset in memory, this is not possible if the data is too large.

- Dask Dataframes divide the Pandas dataframe in smaller ones identified by their index. These new smaller dataframes are called **Partitions;**

- Dask create a graph of these partitions, similarly to what we have seen with dask.delayed. The needed partition will be loaded when it is called, i.e. a **lazy** approach;

Dask Dataframes

We use again the flight dataset we used for Polars:

```
from pathlib import Path
from zipfile import ZipFile
import requests

data_dir = Path("./sample_data") # replace this with a directory of your choice
dest = data_dir / "flights.csv.zip"

if not dest.exists():
    r = requests.get(

"https://transtats.bts.gov/PREZIP/On_Time_Reporting_Carrier_On_Time_Performance_1987_present_2022_1.zip",
        verify=False,
        stream=True,
    )
    data_dir.mkdir(exist_ok=True)
    with dest.open("wb") as f:
        for chunk in r.iter_content(chunk_size=102400):
            if chunk:
                f.write(chunk)

    with ZipFile(dest) as zf:
        zf.extract(zf.filelist[0].filename, path=data_dir)
```

Dask Dataframes: loading data

```
import dask.dataframe as dd
import dask
```

```
ddf = dd.read_csv(extracted, dtype={'ArrTime': 'float64',
       'ArrivalDelayGroups': 'float64',
       'CancellationCode': 'object',
       'DepTime': 'float64',
       'DepartureDelayGroups': 'float64',
       'Div1Airport': 'object',
       'Div1TailNum': 'object',
       'Div2Airport': 'object',
       'Div2TailNum': 'object',
       'WheelsOff': 'float64',
       'WheelsOn': 'float64'})
```

```
ddf.head(3)
```

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	UniqueCarrier	FlightNum	...	AirTime	ArrDelay	DepDelay
0	1999	1	1	5	1526.0	1515	1838.0	1849	CO	1923	...	289.0	-11.0	11.0
1	1999	1	2	6	1727.0	1540	2056.0	1914	CO	1923	...	301.0	102.0	107.0
2	1999	1	3	7	1609.0	1515	1935.0	1849	CO	1923	...	295.0	46.0	54.0

3 rows × 23 columns

Dask Dataframes: loading data

We can also import multiple csv files

```
from glob import glob
import os
```

```
filepath = glob("/content/sample_data/nycflights/*.csv")
```

```
filepath
```

```
['/content/sample_data/nycflights/1994.csv',
 '/content/sample_data/nycflights/1999.csv',
 '/content/sample_data/nycflights/1993.csv',
 '/content/sample_data/nycflights/1990.csv',
 '/content/sample_data/nycflights/1998.csv',
 '/content/sample_data/nycflights/1991.csv',
 '/content/sample_data/nycflights/1996.csv',
 '/content/sample_data/nycflights/1995.csv',
 '/content/sample_data/nycflights/1997.csv',
 '/content/sample_data/nycflights/1992.csv']
```

Dask Dataframes: loading data

We can also import multiple csv files

```
ddf = dd.read_csv(filepath)
```

```
ddf.head(3)
```

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	UniqueCarrier	FlightNum	...	AirTime	ArrDelay	DepDelay
0	1994	1	1	6	1401.0	1400	1454.0	1507	US	241	...	NaN	-13.0	1.0
1	1994	1	2	7	1400.0	1400	1448.0	1507	US	241	...	NaN	-19.0	0.0
2	1994	1	3	1	1400.0	1400	1455.0	1507	US	241	...	NaN	-12.0	0.0

3 rows × 23 columns

```
ddf.tail(3)
```

	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	UniqueCarrier	FlightNum	...	AirTime	ArrDelay	DepDe
262563	1992	12	29	2	2049.0	1944	2347.0	2247	CO	1579	...	NaN	60.0	
262564	1992	12	30	3	1950.0	1944	2255.0	2247	CO	1579	...	NaN	8.0	
262565	1992	12	31	4	1944.0	1944	2232.0	2247	CO	1579	...	NaN	-15.0	

3 rows × 23 columns

Dask Dataframes: computations

If you want to compute the max of the column DepDelay in pandas you have to open all files and save the max:

```
maxes = []
for fn in filepath:
    df = pd.read_csv(fn)
    maxes.append(df["DepDelay"].max())

final_max = max(maxes)
```

We could parallelize all this by wrapping the read_csv with dask_delayed, but it could be bothersome. Dask dataframes let us do these operations directly by using a pandas-like code.

Dask Dataframes: performance

Let's compare Dask and Pandas:

```
%%time
ddf =  dd.read_csv(filepath, dtype={'CRSElapsedTime': 'float64',
       'TailNum': 'object'})
ddf["DepDelay"].max().compute()
```

```
CPU times: user 7.22 s, sys: 913 ms, total: 8.14 s
Wall time: 5.17 s
1435.0
```

```
%%time
pdf = pd.concat(pd.read_csv(f) for f in filepath)
pdf["DepDelay"].max()
```

```
CPU times: user 4.1 s, sys: 185 ms, total: 4.29 s
Wall time: 4.59 s
1435.0
```

Dask Dataframes: performance

Let's compare Dask and Pandas:

```
%%time
ddf =  dd.read_csv(filepath, dtype={'CRSElapsedTime': 'float64',
       'TailNum': 'object'})
ddf["DepDelay"].max().compute()
```

```
CPU times: user 7.22 s, sys: 913 ms, total: 8.14 s
Wall time: 5.17 s
1435.0
```

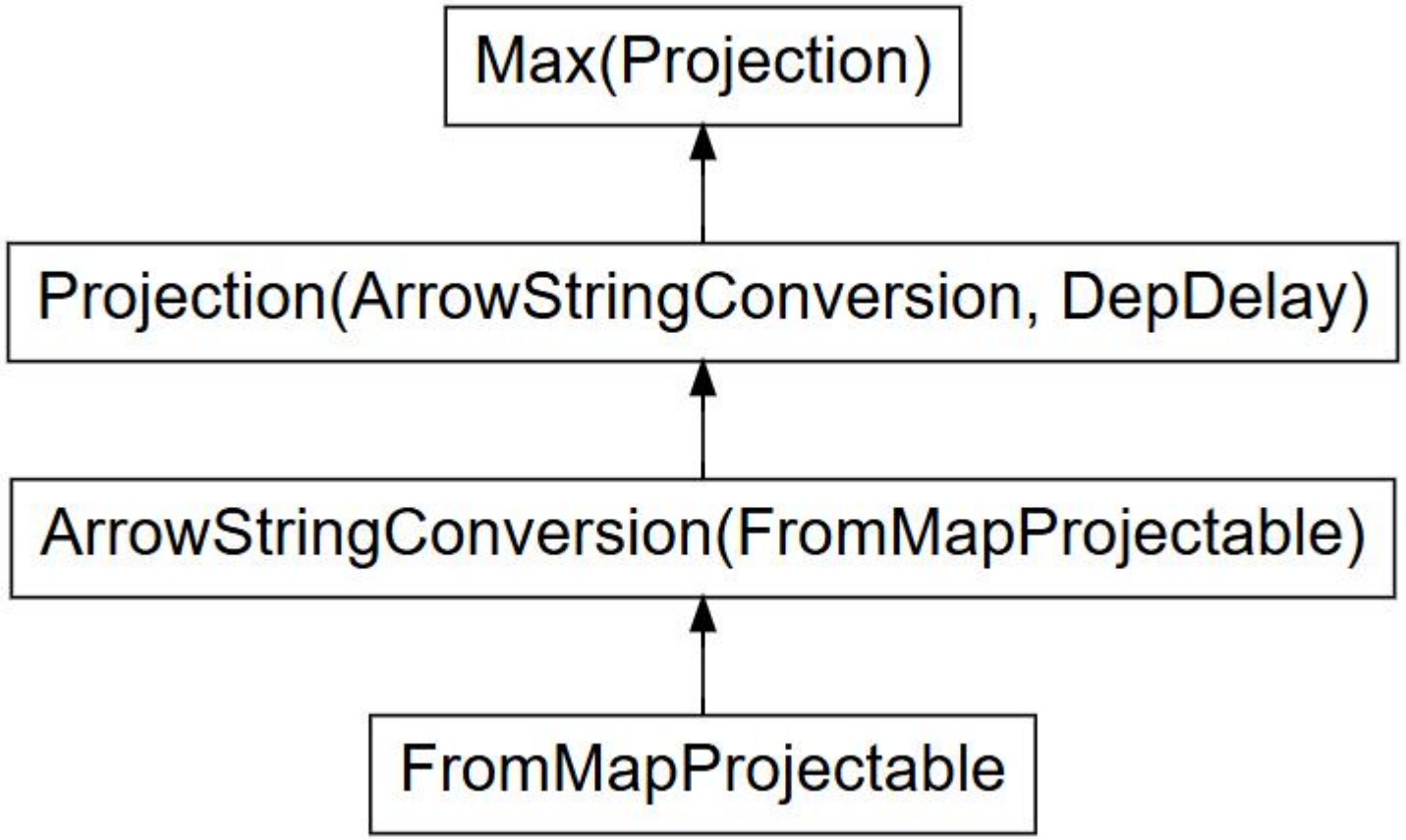
```
%%time
pdf = pd.concat(pd.read_csv(f) for f in filepath)
pdf["DepDelay"].max()
```

```
CPU times: user 4.1 s, sys: 185 ms, total: 4.29 s
Wall time: 4.59 s
1435.0
```

Dask Dataframes: lazy approach

We can view the underlying task graph using the method visualize:

```
ddf.DepDelay.max().visualize()
```



Exercises

1. How many rows are in the dataset? (Hint: use len())
2. How many non-canceled flights were taken?
3. What was the average DepDelay from each airport? (Hint: use ddf.groupby())
4. How many non-cancelled flights were taken from each airport? (Hint: use ddf.groupby())
5. What DayOfWeek has the biggest DepDelay? (Hint: use ddf.groupby())

Solutions

1.
```
len(ddf)
```
```
2611892
```

2.
```
len(ddf[ddf.Cancelled==False])
```
```
2540961
```

3.
```
%time ddf["DepDelay"].groupby(ddf["Origin"]).mean().compute()
```
```
CPU times: user 7.58 s, sys: 678 ms, total: 8.26 s
Wall time: 5.24 s
```

Origin	DepDelay
EWR	10.295469
JFK	10.351299
LGA	7.431142

dtype: float64

Solutions

4.
```
ddf2 = ddf[ddf.Cancelled==False]
ddf2.groupby("Origin").count().compute()
```

Origin	Year	Month	DayofMonth	DayOfWeek	DepTime	CRSDepTime	ArrTime	CRSArrTime	UniqueCarrier	FlightNum	...	CRSElapsedTime	Air
EWR	1139451	1139451	1139451	1139451	1139451	1139451	1136451	1139451	1139451	1139451	...	1139418	57
JFK	427243	427243	427243	427243	427243	427243	425937	427243	427243	427243	...	427219	21
LGA	974267	974267	974267	974267	974267	974267	971294	974267	974267	974267	...	974186	46

3 rows × 22 columns

5.
```
ddf["DepDelay"].groupby(ddf["DayOfWeek"]).mean().compute()
```

DayOfWeek	DepDelay
1	8.096565
2	8.149109
3	9.141912
4	10.538275
5	11.476687
6	7.824071
7	8.994296

dtype: float64

Dask distributed

Until now we have seen some simple way to parallelize our code with Dask. Now we will see how to set up a more complete environment through the creation of a **Dusk cluster**. A Dusk cluster is composed by:

- **Scheduler**: A single, centralized process which manage computation request, the status of tasks and workers and send the tasks;
- **Workers**: one of the processes that compute tasks
- **Clients**: User-facing entry points to interact with the cluster;

Dask distributed: setup

To set up correctly a Dask Cluster you have set up ports through which it will communicate. We won't get deeper in this topic, so just do the minimum:

```
import os
import dask
import dask.config
import dask.distributed
```

```
DASHBOARD_PORT = 45000
SCHEDULER_PORT = 46000
```

```
from dask.distributed import LocalCluster

# Launch a scheduler and 4 workers on my local machine
cluster = LocalCluster(n_workers=4, threads_per_worker=2,
scheduler_port=SCHEDULER_PORT, dashboard_address=DASHBOARD_PORT)
cluster
```

Dask distributed: setup

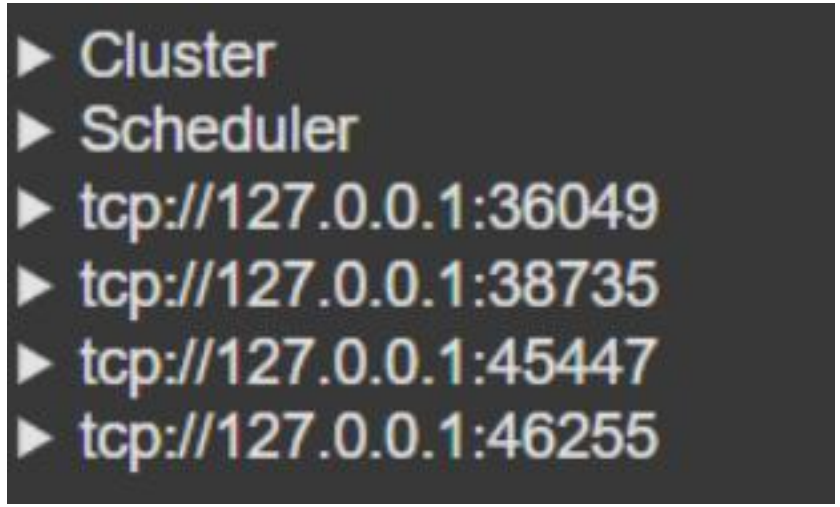
```
from dask.distributed import Client

client = Client(cluster)
```

You can scale up or down the number of workers:

```
cluster.scale()
```

You can get logs:

```
cluster.get_logs()
```



Dask distributed: execution

```
from time import sleep
import numpy as np
from dask import delayed
```

```
def inc(x):
    sleep(1)
    return x + 1

def add(x, y):
    sleep(1)
    return x + y
```

```
%%time
results = []
data = np.array([1,2,3,4])
for x in data:
    y = delayed(inc)(x)
    results.append(y)

total = delayed(sum)(results)
print("Before computing:", total)  # Let's see what total is
result = total.compute()
print("After computing :", result)  # After it's computed
```

```
Before computing: Delayed('sum-2e085a31-
8cc8-48e7-992e-ca6b3a0e3b2e')
After computing : 14
CPU times: user 7.96 ms, sys: 2.95 ms,
total: 10.9 ms
Wall time: 2.01 s
```

Dask distributed: execution

```
import dask.array as da
```

```
x = da.random.random((10_000,10_000,10), chunks=(1000,250,5))
y = da.random.random((10_000,10_000,10), chunks=(1000,250,5))
z = (da.arcsin(x) + da.arccos(y)).sum(axis=(1,))
```

```
%%time
z.compute()
```

```
CPU times: user 15.3 s, sys: 1.19 s, total: 16.5 s
Wall time: 1min 37s
array([[15682.47430494, 15702.61882835, 15707.00326162, ...,
        15732.62083973, 15725.47441854, 15740.75736981],
       [15612.72157425, 15698.41560249, 15757.61946288, ...,
        15622.76389998, 15781.01564013, 15727.29636642],
       [15771.30138821, 15736.24833914, 15643.64842393, ...,
        15734.14871173, 15693.04790217, 15668.67702725],
       ...,
       [15630.41520875, 15669.68589092, 15694.25547628, ...,
        15639.17082433, 15678.89492289, 15799.54460098],
       [15760.54310366, 15700.27909301, 15759.64516009, ...,
        15618.70798503, 15683.19228552, 15738.12884942]])
```

Once declared a client all compute will be performed on it by deafult

Dask distributed: Futures

Futures are non-blocking, distributed computations. They are eager instead of lazy. This means that computation start immediately

```
def inc(x):
    return x + 1

def add(x, y):
    return x + y
```

We can send the function to the cluster in order to be executed with submit

```
a = client.submit(inc, 10)  # calls inc(10) in background process
b = client.submit(inc, 20)
```

Dask distributed: Futures

It will be returned a future object that gives us info on the job status

```
a
```

```
Future: inc status: finished, type: int, key: inc-801750d6c5eec01f724c11c64d2ce46a
```

to get the result:

```
a.result()
```

```
11
```

we can call other futures as input:

```
c = client.submit(add, a, b)
```

```
c.result()
```

```
32
```

Dask distributed: functions on array

We can use client.map to apply a function to arrays.

```
arr = np.arange(1,15,2)
```

```
map_arr = client.map(inc, arr)
```

```
map_arr
```

```
[<Future: finished, type: numpy.int64, key: inc-722159f2ddce14659661b564cb2d064e>,
 <Future: finished, type: numpy.int64, key: inc-bc066b32300d4dc1fe7f8df5f92d2170>,
 <Future: finished, type: numpy.int64, key: inc-1010edff8abf6abc3f78f4f8f701b1d9>,
 <Future: finished, type: numpy.int64, key: inc-3a4d98d03dea265d906b2f46ed735095>,
 <Future: finished, type: numpy.int64, key: inc-07f218800b3ba535028da200ac4d389e>,
 <Future: finished, type: numpy.int64, key: inc-99420eb9d8aa1f41b7e85ca0eb80d9e1>,
 <Future: finished, type: numpy.int64, key: inc-44068dc842ba5de95af4d53b6bbca31e>]
```

Dask distributed: functions on array

We can use gather to get the results:

```
client.gather(map_arr)
```

```
[np.int64(2),
 np.int64(4),
 np.int64(6),
 np.int64(8),
 np.int64(10),
 np.int64(12),
 np.int64(14)]
```

We can also distribute all the futures on the cluster to then get the results:

```
remote_arr = client.scatter(arr)
```

```
func_arr = client.submit(inc, remote_arr)
```

```
func_arr.result()
```

```
array([ 2,  4,  6,  8, 10, 12, 14])
```

Dask distributed: functions on array

This is useful if you problem is memory-bound, in all other cases numpy is still faster:

```
arr_2 = np.arange(0, 10_000,1).reshape(100,100)
```

```
%%time
np.sin(arr_2)
```

```
CPU times: user 1.62 ms, sys: 0 ns, total: 1.62 ms
Wall time: 4.57 ms
```

```
%%time
future = client.map(np.sin,arr_2)
client.gather(future)
```

```
CPU times: user 159 ms, sys: 15.7 ms, total: 175 ms
Wall time: 255 ms
```

```
cluster.close()
client.shutdown()
```

Dask distributed: GPU cluster

We can also spin up a cluster able to use the Nvidia GPUs present in our system:

```
from dask_cuda import LocalCUDACluster

cluster = LocalCUDACluster()
cluster
```

```
client = Client(cluster)
```

You can have a single worker for each GPU. The package dask_cuda makes sure it is so with similar commands as the standard dask. It inspects the hardware available and starts one worker per GPU, setting option in order that each one can run only on their respective GPU.

Dask distributed: GPU cluster

Let’s try a kernel

```
from numba import cuda

@cuda.jit
def some_kernel():
    i = 0
    while i < 1_000_000:
        i += 1

f = client.submit(some_kernel[1024*1024, 1024])
```

```
f
```

```
Future: _LaunchConfiguration status: finished, type: NoneType, key:
<numba.cuda.dispatcher._LaunchConfiguration object-dc00d9fdac294b1835a8de4bf50e795b
```

THANKS!

IR0000032 – ITINERIS, Italian Integrated Environmental Research Infrastructures System
(D.D. n. 130/2022 - CUP B53C22002150006) Funded by EU - Next Generation EU PNRR-Mission 4 “Education and Research” - Component 2: “From research to business” - Investment 3.1: “Fund for the realisation of an integrated system of research and innovation infrastructures”

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani