From Local Dask to Distributed Dask

Tommaso Tedeschi
tommaso.tedeschi@pg.infn.it

ICSC
Centro Nazionale di Ricerca in HPC,
Big Data and Quantum Computing

INFN
PERUGIA
Istituto Nazionale di Fisica Nucleare
Sezione di Perugia

Single machine vs multiple machines

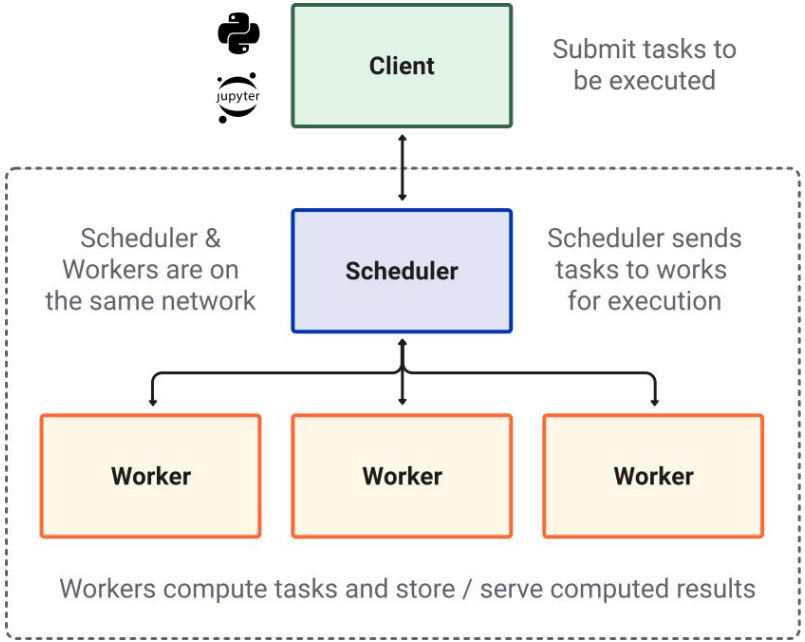
As we have seen, Dask runs perfectly well on a single machine with or without a distributed scheduler. But once you start using Dask in anger you'll find a lot of benefit both in terms of scaling and debugging by using the distributed scheduler.

- Default (single-machine) Scheduler
  - The no-setup default. Uses local threads or processes for larger-than-memory processing
- Dask.distributed
  - The sophistication of the newer system on a single machine. This provides more advanced features while still requiring almost no setup.

Dask.distributed

https://distributed.dask.org/en/stable/

- **Dask.distributed is a centrally managed, distributed, dynamic task scheduler**:
  - The central **dask scheduler** process coordinates the actions of several **dask worker** processes spread across multiple machines and the concurrent requests of several clients.
  - a worker is a Python object and node in a dask Cluster that performs computations and serves computed results
- Users interact by connecting a local Python session to the scheduler and submitting work, via `client.submit(function, *args, **kwargs)` or by using the large data collections and parallel algorithms of the parent dask library.
  - the simple `client.submit` interface provides users with custom control to submit fully custom workloads.



Why dask.distributed?

Dask.distributed meets the following needs:

- **Low latency**: 1ms task overhead
- **Peer-to-peer data sharing**: Workers communicate with each other to share data
- **Complex Scheduling**: Supports complex workflows (not just map/filter/reduce)
- **Pure Python**: Built in Python using well-known technologies
- **Data Locality**: Scheduling algorithms cleverly execute computations where data lives
- **Familiar APIs**: Compatible with the concurrent.futures API in the Python standard library and dask API for parallel algorithms
- **Easy Setup**: As a Pure Python package distributed is pip installable and easy to set up on your own cluster.

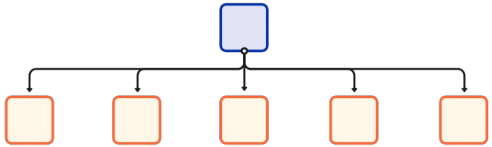
Cluster Manager class

Best way to start the distributed scheduler and worker components way is to use a **cluster manager utility class**

- These cluster managers deploy a scheduler and the necessary workers as determined by communicating with the **resource manager**
- All cluster managers follow the same interface, but with platform-specific configuration options
  - you can switch from your local machine to a remote cluster with very minimal code changes

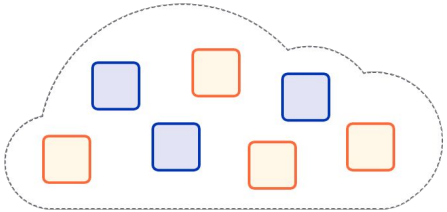
Cluster Manager

Deploys one Scheduler and many Workers by talking to the Resource Manager



Resource Manager

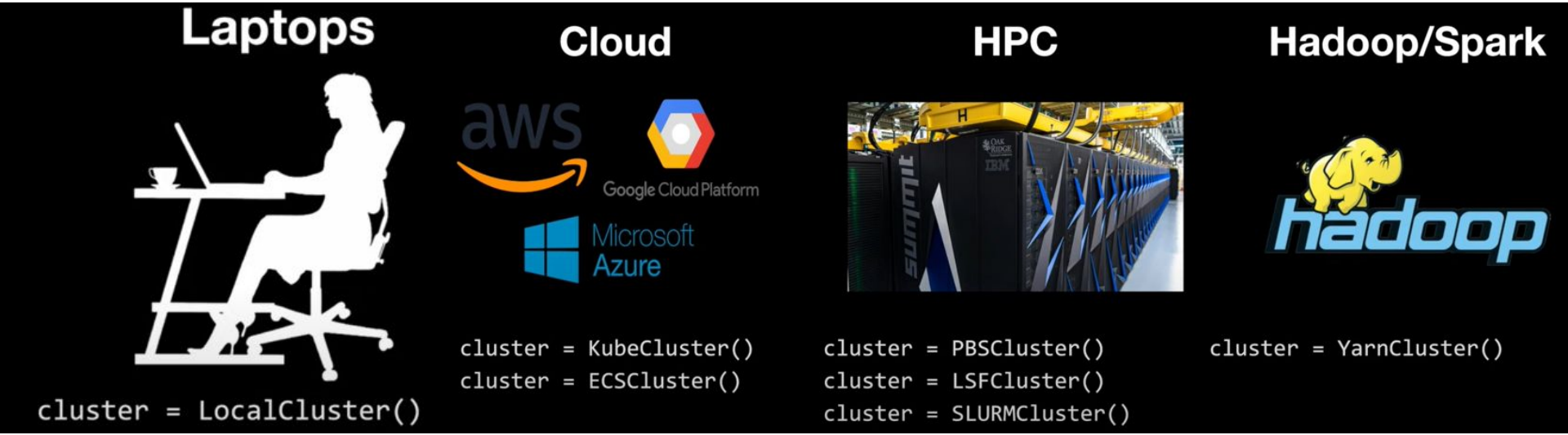
Kubernetes/Yarn/SLURM/PBS/Abstract pods/jobs on top of Physical Hardware



Physical Hardware

Physical CPUs, GPUs, networking and storage; either on-prem or on the cloud

Available Cluster Managers



When we instantiate a `Client()` object with no arguments it will attempt to locate a Dask cluster. It will check your local Dask config and environment variables to see if connection information has been specified. If not it will create an instance of LocalCluster and use that

Dask-Jobqueue

Dask-jobqueue is a set of cluster managers for HPC users and works with job queueing systems

https://jobqueue.dask.org/en/latest/

- Supports PBS, Slurm, SGE and HTCondor
  - typically found in high performance supercomputers, academic research institutions, and other clusters
- Provides a convenient interface that is accessible from interactive systems like Jupyter notebooks, or batch jobs.
- Creates a Dask Scheduler in the Python process where the cluster object is instantiated
- The cluster generates a traditional job script and submits that an appropriate number of times to the job queue
- Jobs are resources submitted to, and managed by, the job queueing system:
  - a single Job may include one or more Workers

```
from dask_jobqueue import PBSCluster
cluster = PBSCluster()
cluster.scale(jobs=10)    # Deploy ten single-node jobs

from dask.distributed import Client
client = Client(cluster)  # Connect this local process to remote workers

# wait for jobs to arrive, depending on the queue, this may take some time

import dask.array as da
x = ...                   # Dask commands now use these distributed resources
```

Interactive dashboard

Dask's distributed scheduler provides an **interactive dashboard** to live monitoring of your Dask computations.

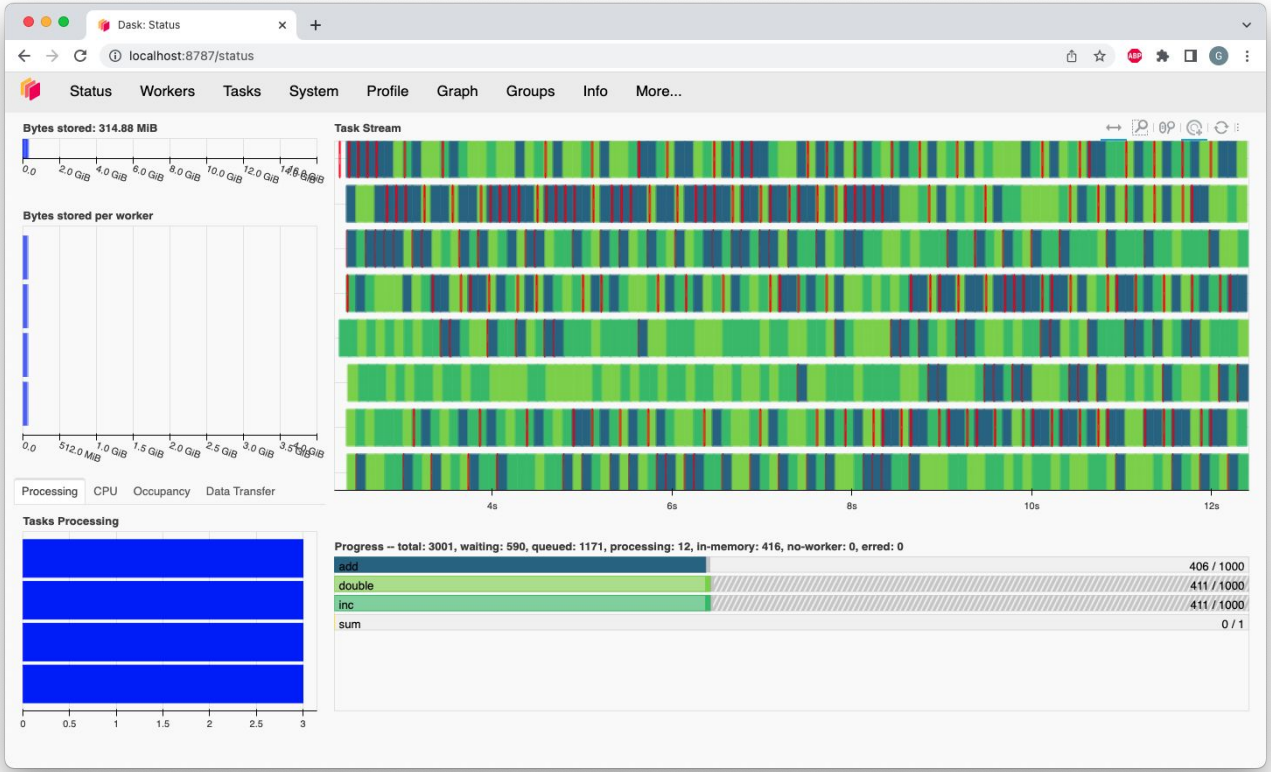
- The dashboard is built with Bokeh and will start up automatically, returning a link to the dashboard whenever the scheduler is created.

There are numerous diagnostic plots available:

- Bytes Stored and Bytes per Worker:
  - Cluster memory and Memory per worker
- Task Processing/CPU Utilization/Occupancy/Data Transfer:
  - Tasks being processed by each worker/ CPU Utilization per worker/ Expected runtime for all tasks currently on a worker.
- Task Stream:
  - Individual task across threads.
- Progress:
  - Progress of a set of tasks.

https://docs.dask.org/en/latest/dashboard.html

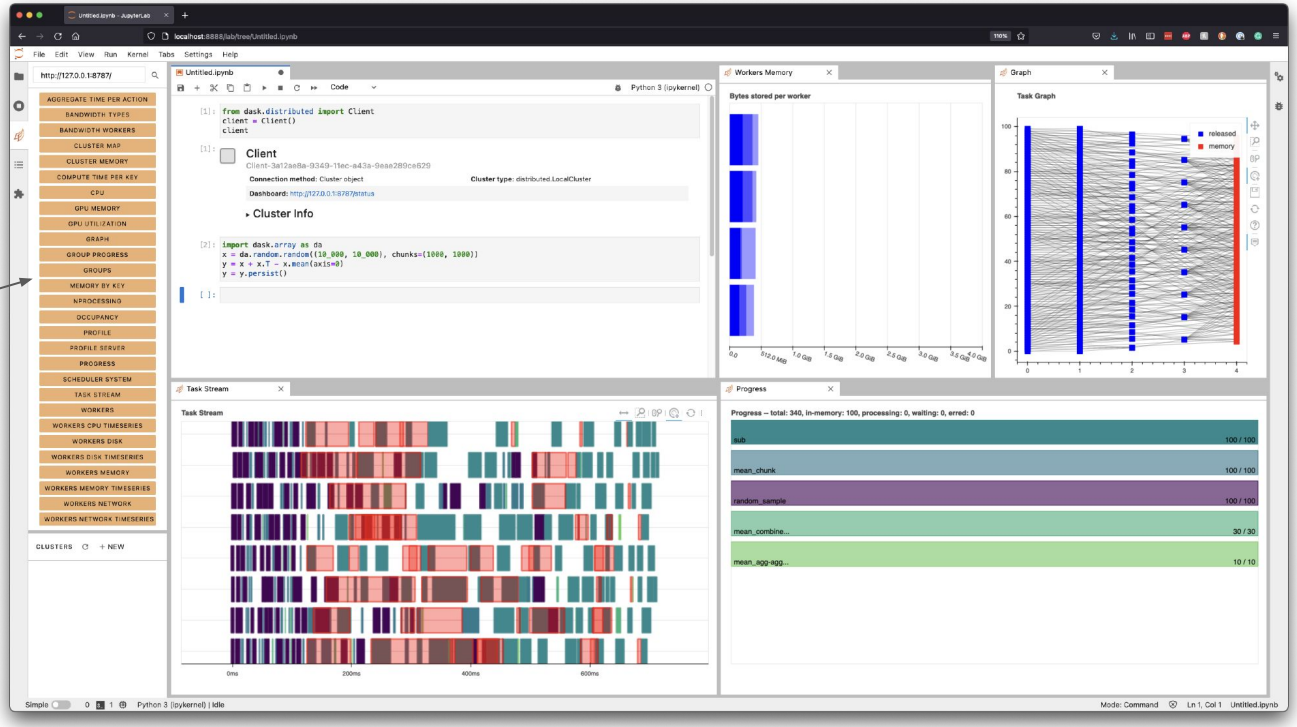
Interactive dashboard



The JupyterLab extension

The **JupyterLab Dask extension** allows you to embed Dask's dashboard plots directly into JupyterLab panes.

Once the JupyterLab Dask extension is installed you can choose any of the individual plots available and integrated as a pane in your JupyterLab session.



Next-gen HEP data analysis

Two main analysis tools arising:

- ROOT's RDataFrame
- HSF's Coffea (based on awkward array + uproot libraries)

They are both thought to use different backends, including Dask

Dask capability to be deployed on batch/distributed resources (Dask-Jobqueue and Dask-Kubernetes) is a game-changer in the utilization of legacy resources:

- **Coffea/RDataFrame + Dask** is emerging as the "standard" solution for the current/future high-throughput analysis

```
import ROOT
from dask.distributed import Client

# Point RDataFrame calls to the Dask specific RDataFrame
RDataFrame = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame

# In a Python script the Dask client needs to be initalized in a context
# Jupyter notebooks / Python session don't need this
if __name__ == "__main__":
    # With an already setup cluster that exposes a Dask scheduler endpoint
    client = Client("dask_scheduler.domain.com:8786")

    # The Dask RDataFrame constructor accepts the Dask Client object as an optional argument
    df = RDataFrame("mytree","myfile.root", daskclient=client)
    # Proceed as usual
    df.Define("x","someoperation").Histo1D(("name", "title", 10, 0, 10), "x")
```

```
import coffea.processor as processor
import awkward as ak
from coffea.nanoevents import schemas

from dask.distributed import Client

client = Client("tls://localhost:8786")

fileset = {'SingleMu' : ["root://eospublic.cern.ch//eos/root-eos/benchmark/Run2012B_SingleMu.root"]}

executor = processor.DaskExecutor(client=client)

run = processor.Runner(executor=executor,
                       schema=schemas.NanoAODSchema,
                       savemetrics=True
                      )

output, metrics = run(fileset, "Events", processor_instance=Processor())
```

Example - RDataFrame + Dask

What happens under the hood in Distributed RDataFrame on Dask?

- Dask delayed is used to create a map-reduce task graph

- then results are computed and persisted persisted

https://github.com/root-project/root/blob/master/bindings/experimental/distrdf/python/DistRDF/Backends/Dask/Backend.py

```
def ProcessAndMerge(self,
                    ranges: List[Any],
                    mapper: Callable[[Ranges.DataRange,
                                      Callable[[Union[Ranges.EmptySourceRange, Ranges.TreeRangePerc]],
                                               Base.TaskObjects],
                                      Callable[[ROOT.RDF.RNode, int], List],
                                      Callable],
                                      Base.TaskResult],
                    reducer: Callable[[Base.TaskResult, Base.TaskResult], Base.TaskResult],
                    ) -> Base.TaskResult:
    """
    Performs map-reduce using Dask framework.

    Args:
        ranges (list): A list of ranges to be processed.
        mapper (function): A function that runs the computational graph
            and returns a list of values.

        reducer (function): A function that merges two lists that were
            returned by the mapper.

    Returns:
        list: A list representing the values of action nodes returned
        after computation (Map-Reduce).
    """
    dmapper = dask.delayed(DaskBackend.dask_mapper)
    dreducer = dask.delayed(reducer)

    mergeables_lists = [dmapper(range, self.headers, self.shared_libraries, mapper) for range in ranges]

    while len(mergeables_lists) > 1:
        mergeables_lists.append(
            dreducer(mergeables_lists.pop(0), mergeables_lists.pop(0)))

    # Here we start the progressbar for the current RDF computation graph
    # running on the Dask client. This expects a future object, so we need
    # convert the last delayed object from the list above to a future
    # through the `persist` call. This also starts the computation in the
    # background, but the time difference is negligible. The progressbar is
    # properly shown in the terminal, whereas in the notebook it can be
    # shown only if it's the last call in a cell. Since we're encapsulating
    # it in this class, it won't be shown. Full details at
    # https://docs.dask.org/en/latest/diagnostics-distributed.html#dask.distributed.progress
    final_results = mergeables_lists.pop().persist()
    progress(final_results)

    return final_results.compute()
```

Future analysis at LHC

R&D on analysis at High Luminosity LHC (HL-LHC)

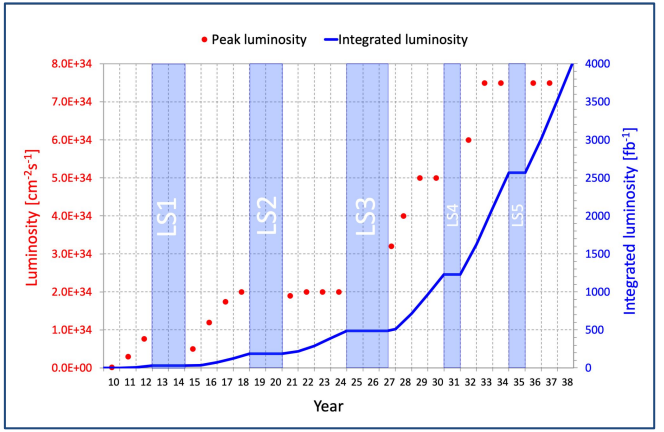
- Promote adoption of reduced data formats
- optimizing the computing and storage resource utilization

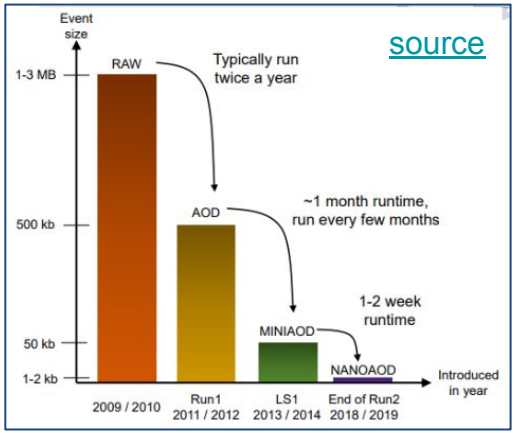
Testing software featuring a declarative programming model and interactive workflows

- Increasing data processing throughput is crucial
- Ergonomic interfaces remove the lower-level programming burden from analysts
- Fast Turnaround Reducing analysis “time to insight”

Prototype resources integration models to efficiently leverage computing capacity

- Integrate already deployed (grid) infrastructure
- Transparently access specialized HW
- Scale toward opportunistic (cloud/HPC)



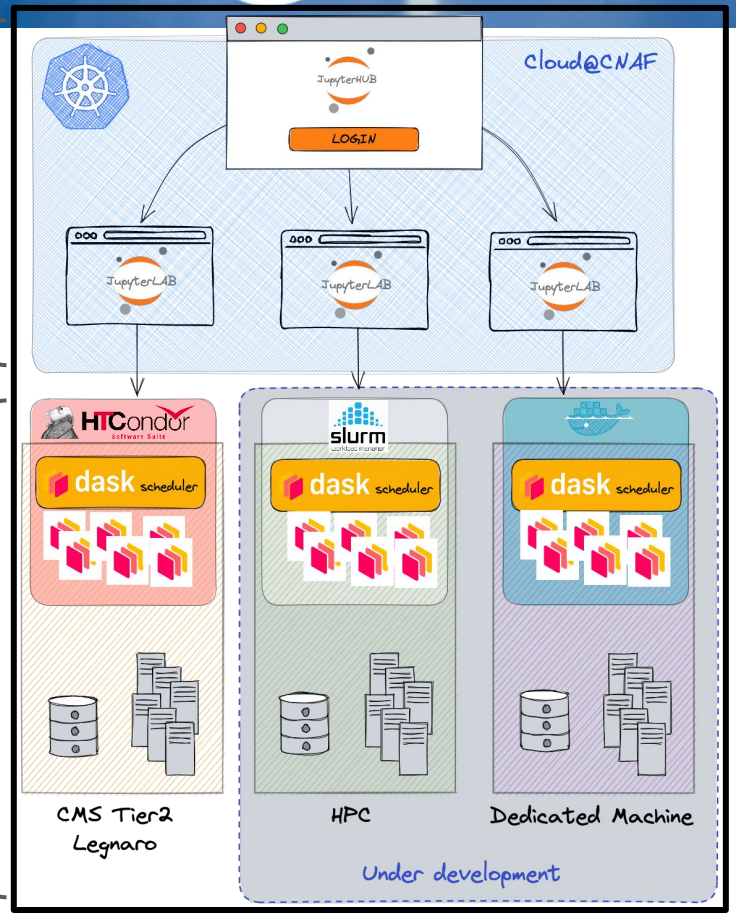


The INFN solution

Interactivity is reached via Dask distributed, deployed on existing HTCondor distributed resources via a custom version of Dask-jobqueue https://github.com/comp-dev-cms-ita/dask-remote-jobqueue

What users see

What the offloading hides to the user



Instructions
https://infn-cms-analysisfacility.readthedocs.io/en/latest/

Let’s have a look

- To dive deeper into Dask distributed:
  - https://tutorial.dask.org/04_distributed.html
- An example of RDataFrame + Dask computation:
  - https://github.com/comp-dev-cms-ita/pyHEP2022_distRDF_INFN_AF

Screencast example:
https://drive.google.com/file/d/1Sgn9rZiKXKeUgYUt2w-oOv3ZCTE17Zbz/view?usp=share_link