Execution of a feature engineering algorithm in Python on a supercomputer with Dask
Context

- Introduction to the problematic: derivation
- Dask
- Derivation toy model
- Chaining futures

Dask on Myria

- CRIANN Presentation
- Supercomputer *Myria*
- Dask on Myria

Conclusion
Introduction to the problematic: derivation
Primary data used in our data pipeline

Example: ESG scores

<table>
<thead>
<tr>
<th>Share 1</th>
<th>Share 2</th>
<th>Share 3</th>
<th>Share 4</th>
<th>Share 5</th>
<th>Share 6</th>
<th>Share 7</th>
<th>Share 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-09-30</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2010-10-01</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2010-10-04</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2010-10-05</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2010-10-06</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2010-10-07</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2010-10-08</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2010-10-11</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2010-10-12</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10 years (2600 business days)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x276 of such dataframes (scores, prices, sectors, marketcap ...) > 700x10^6 cells
Feature engineering

A set of transformations/combinations called ”derivations” is applied to the primary data in order to feed the learnings.
Ex. : volatility & returns in sectors
Ex.: compute only the returns

- Primary data: prices, sectors
- Calculations: pctdelta, returns
- Output: volatility, clusterisation, returns per sector

What is Dask?

Dask

• Python library for parallel data processing
• Fully integrated in the PyData Stack (NumPy, SciPy, Scikit-Learn, Pandas, ...)

Main parts

1. Dynamic task scheduling optimized for computation
2. Big Data collections for parallel data management

How dask work?

• Processing instructions on data generates a task graph
• The graph is scheduled by a cluster of workers
• Several solutions to manage the cluster: python multiprocessing (for single node only), k8s cluster, job scheduler like Slurm or PBS, MPI communicator, ...
Operators

Unary operator

- moving
- average

generic operator

- correl.

Map

- moving
- average
Derivation toy model: full graph

Prices → pctdelta → ewmstd → get cluster → calc_dev_by → calc_dev_by → calc_dev_by

Zscore → Prices

Sectors
```python
class Basic(AbstractDerivation):
    """The basic class to read data from data base""
    all_prices = FileVar('all_prices.csv')
    marketcap = FileVar('marketcap.csv')
    sectors_isin = FileVar('Sectors.csv')

class Temporary(AbstractDerivation):
    """Temporary data for compute Y and X and others for the trading""
    fin_rdt_1d_all = Var(op.pctdelta, Basic.all_prices, {'period': 1})
    fin_rdt_1d_std = Var(op.ewmstd, fin_rdt_1d_all, {'span': 260, 'min_periods': 260, 'annulize': True})
    clustervol = Var(op.get_clusters, fin_rdt_1d_std, {'levels': 5, 'dend': DATE_END_CLUSTERIZATION})
    fin_pricez1d = Var(op.zscore, Basic.all_prices, {'h': 1, 'minp': 60, 'inpct': True})
    fin_pricedevz1d_byvol = Var(op.calc_dev_by, [fin_pricez1d, clustervol],
                               {'do_reduce': False, 'clusterishito': True})
    fin_std260d_Z1d_bysectors = Var(op.calc_dev_by, [fin_pricez1d, Basic.sectors_isin],
                                     {'do_reduce': False})
    fin_pricedevz1d_byuniver = Var(op.calc_dev_by, [fin_pricez1d], {'do_reduce': False})
    a = Var(op.wait_for_dask, fin_pricez1d, {})
```
Chaining futures

```python
def _build_result(self):
    parent_inputs = list(map(self.engine.var_to_input, self.parent_vars))
    self._result.obj = self.engine.submit(self.name, self.operator, parent_inputs, self.params)
```

```python
def __call__(self):
    assert self.name, 'name should be set first from the derivation class'
    logger.info(f'Calling {self.name}')
    assert self._result is not None, 'self._result has not been built'
    if self._result.empty():
        self._build_result()
    return self._result
```
Chaining futures: \_build\_result

```python
def \_build\_result(self):
    parent_inputs = list(map(self.engine.var_to_input, self.parent_vars))
    self._result.obj = self.engine.submit(self.name, self.operator, parent_inputs, self.params)
```

```python
def __call__(self):
    for var_name, var_obj in self.__get_vars():
        if var_obj.persist_path is None and var_name not in self.vars_to_launch:
            continue
        if isinstance(var_obj, LazyMemoizer):
            var_obj()
        var_obj.persist(self.persist_path)
```

```python
def __call__(self):
    assert self.name, 'name should be set first from the derivation class'
    logger.info(f'Calling {self.name}')
    assert self._result is not None, 'self._result has not been built'
    if self._result.is_empty():
        self._build_result()
    return self._result
```
Chaining futures: _build_result

```python
def _build_result(self):
    parent_inputs = list(map(self.engine.var_to_input, self.parent.vars))
    self._result.obj = self.engine.submit(self.name, self.operator, parent_inputs, self.params)
```
Chaining futures: run a graph subset

```python
def _build_result(self):
    parent_inputs = list(map(self.engine.var_to_input, self.parent_vars))
    self._result.obj = self.engine.submit(self.name, self.operator, parent_inputs, self.params)
```

```
def __call__(self):
    for var_name, var_obj in self._get_vars():
        if self.vars_to_launch is not None and var_name not in self.vars_to_launch:
            continue
        if isinstance(var_obj, LazyMemoizer):
            var_obj()
            var_obj.persist(self.persist_path)
```

```
def __call__(self):
    assert self.name, 'name should be set first from the derivation class'
    logger.info(f'Calling {self.name}')
    assert self._result is not None, 'self._result has not been built'
    if self._result.is_empty():
        self._build_result()
    return self._result
```
Chaining futures: feature calls its parents

def _build_result(self):
    parent_inputs = list(map(self.engine.var_to_input, self.parent-vars))
    self._result_obj = self.engine.submit(self.name, self, self.operator, parent_inputs, self.params)

Calls all its parents before "submit"

LazyMemoizer
- name : str
- result : pd.DataFrame
- engine : dask engine
  - build_result
  - persist
  - _call

Derivation
- persist_path : str
- vars : List[LazyMemoizer]
- vars_to_launch : List[str]
  - __init__
  - __call__

Var
- operator : Callable
- parents_vars : List[LazyMemoizer]

FileVar
- path : str

def __call__(self):
    for var_name, var_obj in self.__get_vars():
        if self.vars_to_launch is not None and var_name not in self.vars_to_launch:
            continue
        if isinstance(var_obj, LazyMemoizer):
            var_obj()
            var_obj.persist(self.persist_path)

    if self._result is not None,
    self._result has not been built
    return self._result
Derivation toy model: subset graph

Prices --> pctdelta --> ewmstd --> get cluster

Sectors

Zscore

calls

calc_dev_by

calc_dev_by

calc_dev_by
Derivation toy model: subset graph

Prices

Zscore

calc_dev_by
Real derivation graph subset

View execution
Real derivation graph subset
Normandy Regional Computing Center

- Created in 1991 as a Non Profit Organisation by Higher Educations Institutions & Universities
  - HPC center and regional network
- Located in Rouen Engineering Campus
- 13 employees (9 engineers or PhD)
- Funding is mainly public
  - Running costs supported by Normandy Region
  - Projects / investments funded by projects with EU, French State & Normandy Region
  - A small self-financing part
- HPC for public research, also open to private
Supercomputer *Myria*

**Performance**

- **773 TFlop/s (Peak)**
  - 419 Xeon + 327 GPU + 27 KNL

**Compute nodes**

- **366 Broadwell Nodes**
  - 10248 cores
  - 28 cores@2.4 GHz - 128 GB RAM
- **Specialised nodes**
  - 26 GPUs (48 K80, 17 P100, 20 V100)
  - 13 I/O
- **SMP node Haswell**
  - 256 cores@2.2 GHz
  - 4 To RAM DDR4

**Other Components**

- **10 Xeon Phi KNL** (640 cores)
- **Intel OmniPath**
  - 100 Gbit/s
- **DDN Storage**
  - 2.5 Po (HDD)

**Access**

- 4x10Gbit/s & 1x40Gbit/s
  - 5 front-end & 3 visu. nodes

**Operating Systems**

- Cent OS - Slurm - GPFS

Dask on Myria: History

Since 2020

- Criann context: experimental works for growing competence
- Goal: service available for the next production machine (2023)

2020: 1 user project

- Techno watch for project users (workflow full redesign)
- Distributed data processing
- Mainly dask.dataframe features
- Results: efficient on a single node; dataframe partitioning inefficient in multinode

2022: Advestis

- Production code using Dask
- Distributed data processing
- Mainly dask.distributed features
Dask on Myria

Versions
- available only for privileged users
- specific version for Advestis
  - restricted access
  - in-house packages
- several versions for experimentation
  - Criann only

Dashboard unavailable
web access prohibited
Cluster technologies

LocalCluster

- Uses Python multiprocessing/multithreading to deploy dask workers

```python
1 cluster = LocalCluster()
2 client = Client(cluster)
```

- Pros
  - Easy to setup
  - In a slurm job, automatically adapt workers with the cgroups resources

- Cons
  - For single node only: no scalability

K8S Cluster

- Uses K8S cluster to deploy dask workers

```python
1 from dask_kubernetes import KubeCluster
2 cluster = KubeCluster.from_dict(self.get_dask_kube_worker_spec())
3 cluster.adapt(minimum=1, maximum=self.dask_kube_max_pods)
4 client = Client(cluster)
```

- Pros
  - Easy to scale up
  - Standard approach for Advestis

- Cons
  - Unavailable on Myria (Slurm cluster without container)
Cluster technologies

Job scheduler

- Uses job scheduler (Slurm, PBS, ...) to deploy dask workers
- Configuration file (*jobqueue.yaml*) specific to the cluster
- Pros
  - Fully integrated with HPC clusters
  - Easy to scale up
- Cons
  - Dask workers run in jobs independent of the main Python process: if resources are busy the main process can wait a long time
  - Slurm jobs submitted by slurm jobs: beware side effects

```
jobqueue.yaml

```

```
from dask_jobqueue import SLURMCluster
cluster = SLURMCluster()
```
Cluster technologies

MPI Cluster (dask_mpi)
- Uses MPI to deploy dask workers
- Workers are MPI processes
- Two ways:
  - MPI for main Python process and dask workers
  - MPI only for dask workers
- Pros
  - Easy to scale up
  - Adapted for HPC clusters
- Cons
  - Dysfunctions with real use case (import error, unknown variables)

same MPI context
```python
from dask_mpi import initialize
initialize()
from dask.distributed import Client
client = Client()
```

Warning: n - 2 workers

MPI for dask workers only
```bash
mpirun -np 4 python my_client_script.py
```

Can run in a heterogenous job
```bash
mpirun -np 4 dask-mpi --scheduler-file scheduler.json
```
```python
from dask.distributed import Client
client = Client(scheduler_file='./scheduler.json')
```
Conclusion

Computation graph ran on

- Local machine (Intel Core i9-9900K CPU @ 3.60GHz × 16, 32 GB RAM)
- Kubernetes on a Google Cloud Platform cluster (up to 50 workers, one worker is a 16-core VM)
- Myria

<table>
<thead>
<tr>
<th></th>
<th>full derivation (5327)</th>
<th>medium derivation (1383)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local machine - Memory</td>
<td>120 minutes</td>
<td>20 minutes</td>
</tr>
<tr>
<td>Local machine - Dask</td>
<td>30 minutes</td>
<td>5 minutes</td>
</tr>
<tr>
<td>K8s on GCP - Dask</td>
<td>9 minutes</td>
<td>4 minutes</td>
</tr>
<tr>
<td>Myria - Memory</td>
<td>155 minutes</td>
<td>7 minutes</td>
</tr>
<tr>
<td>Myria - Dask</td>
<td>37 minutes</td>
<td>3 minutes</td>
</tr>
</tbody>
</table>

Computation time
Thank you!

nboumlaik@advestis.com
pcotte@advestis.com
benoist.gaston@criann.fr
Backup slides
With a graph of +1000 nodes:

- How to automatically cache intermediate results to avoid running same nodes multiple times?
- How to run a subset of graph by selecting a subset of final nodes?

⇒ Dask future or Dask delayed?
Dask Delayed

Pros:

- Visualize is pretty cool
- Delayed are lazy objects. It means the complete graph may be configured but not executed. Thus it is possible to trigger a subset of this graph.

Cons:

- Persistence has to be implemented explicitly. For complex graphs it is not sufficient.
Dask Future

Pros:
- persistence happens as long as there is at least one python object referencing the future.

Cons:
- Future is triggered without delay
- There is no visualize method

Conclusion:
- Dask primary components are not sufficient. Thus a thin wrapper is mandatory in order to manage a lazy behavior
- We opted for Dask Futures because they automatically persist
```python
class AbstractDerivation(ABC):
    def __init__(self, persist_folder, vars_to_launch_file=None, fs=None):
        self.persist_path = settings.DERIVED_PATH / persist_folder
        self.vars_to_launch = None
        if isinstance(vars_to_launch_file, Path):
            self.vars_to_launch = vars_to_launch_file.read()
        elif vars_to_launch_file is not None:
            if not isinstance(vars_to_launch_file, list):
                vars_to_launch_file = Path(vars_to_launch_file, fs=fs)
            self.vars_to_launch = vars_to_launch_file.read()
        else:
            self.vars_to_launch = vars_to_launch_file
        if isinstance(self.vars_to_launch, str):
            self.vars_to_launch = self.vars_to_launch.split("\n")
        self.__add_name()
```
```python
def __get_vars(self):
    for var_name in filter(lambda x: not x.startswith('_'), dir(self)):
        var_obj = getattr(self, var_name)
        yield var_name, var_obj

def __add_name(self):
    for var_name, var_obj in self.__get_vars():
        if any(map(lambda x: isinstance(var_obj, x), [Var, Map, FileVar])):
            logger.debug(f'adding name to {var_name}')
            var_obj.name = var_name

def __call__(self):
    for var_name, var_obj in self.__get_vars():
        if self.vars_to_launch is not None and var_name not in self.vars_to_launch:
            continue
        if isinstance(var_obj, LazyMemoizer):
            var_obj()
        var_obj.persist(self.persist_path)
```
class LazyMemoizer(ABC):
    """
    Build a callable object and store the result through an engine
    """

    def __init__(self, *args, **kwargs):
        self._result = None
        self._writer = None

    def __call__(self):
        assert self.name, 'name should be set first from the derivation class'
        logger.info(f'Calling {self.name}')
        assert self._result is not None, 'self._result has not been built'
        if self._result.is_empty():
            self._build_result()
        return self._result
class FileVar(LazyMemoizer):
    
    """
    Setup a Var object from a dataframe stored as a csv file. It is inherited from Var class
    """
    def __init__(self, filename, reader=Reader, folder=settings.BASE_PATH):
        """
        params: filename: used be the reader to get the file
        """
        super().__init__()
        self._result = Result()
        self.reader = reader(folder)
        self.filename = filename
        self.__name = None

@property
    def name(self):
        return self.__name

@name.setter
    def name(self, value):
        self.__name = value

def _build_result(self):
    self._result.obj = self.engine.submit(self.name, self.reader, dfs=(),
                                            params={'name': self.filename})
class Var(LazyMemoizer):
    """
    Setup a var as a transformation from another var
    """

def __init__(self, operator, parent_vars=(), params=None):
    super().__init__()
    self._result = Result()
    assert callable(operator)
    self.operator = operator
    if isinstance(parent_vars, Iterable):
        self.parent_vars = parent_vars
    else:
        self.parent_vars = [parent_vars]
    self.params = params if params else {}
    self.__name = None

    def _build_result(self):
        parent_inputs = list(map(self.engine.var_to_input, self.parent_vars))
        self._result.obj = self.engine.submit(self.name, self.operator, parent_inputs, self.params)

    def persist(self, path):
        writer = Writer(path)
        self._writer = Var(writer, self, params={name: self.name})
        self._writer.name = f'writer_{{self.name}}'
        self._writer()
```
class Basic(AbstractDerivation):
    """The basic class to read data from data base""
    all_prices = FileVar('all_prices.csv')
    marketcap = FileVar('marketcap.csv')
    sectors_isin = FileVar('Sectors.csv')

class Temporary(AbstractDerivation):
    """Temporary data for compute Y and X and others for the trading""
    fin_rdt_1d_all = Var(op.pctdelta, Basic.all_prices, {'period': 1})
    fin_rdt_1d_std = Var(op.ewmstd, fin_rdt_1d_all, {'span': 260, 'min_periods': 260,
               'annulize': True})

    clustervol = Var(op.get_clusters, fin_rdt_1d_std, {'levels': 5,
               'dend': DATE_END_CLUSTERIZATION})

    fin_pricez1d = Var(op.zscore, Basic.all_prices, {'h': 1, 'minp': 60, 'inpct': True})
    fin_pricedevz1d_byvol = Var(op.calc_dev_by, [fin_pricez1d, clustervol],
               {'do_reduce': False, 'clusterishito': True})
    fin_std260d_z1d_bysectors = Var(op.calc_dev_by, [fin_pricez1d, Basic.sectors_isin],
               {'do_reduce': False})
    fin_pricedevz1d_byuniver = Var(op.calc_dev_by, [fin_pricez1d], {'do_reduce': False})
    a = Var(op.wait_for_dask, fin_pricez1d, {})```
def main(vars_to_launch=None, fs=None):
    make_directories()
    derivations = [Basic(settings.S_FOLDER),
                   Temporary(settings.T_FOLDER)]
    for d in derivations:
        d()
    get_engine().wait()
import click

from adoptim3.derivation.append import update_db
# from adoptim3.derivation import deriv_ia_stoxx600
from adoptim3.derivation import deriv_toy_model
# from adoptim3.derivation import deriv_darling

@click.group()
def cli():
    # override me
    pass

@click.command()
@click.option("-l", "--vars_to_launch", default=None)
@click.option("-f", "--fs", default=None)
def derivtoymodel(vars_to_launch, fs):
    deriv_toy_model.main(vars_to_launch, fs)

cli.add_command(derivtoymodel)

if __name__ == '__main__':
    cli()