INTRODUCTION TO HPC WORKFLOWS AS A SERVICE AND SOFTWARE STACK (Session 2)

14th September 2022

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union’s Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

Outline

- **Session 2: Other Software Components (15 min each +5 mins questions)**
  - EDDL for ML in Project Pillars
  - Ophidia for HPDA in Project Pillars
  - dataClay split

Part 1: EDDL for ML in Project Pillars

Jose Filch (UPV)

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

What is EDDL

- **EDDL: European Distributed Deep Learning Library**
- **Open Source library (available on GitHub)**
- **Enables definition, training and inference of neural network models**
    - Written in C++
    - Multi-device support: CPU, GPU, FPGA
    - Tensor operations support
    - Distributed training support
- **Abstracts away infrastructure complexity**
- **pyEDDL: python wrapper**
    - OpenSource (available on GitHub)
- **Both developed in the framework of DeepHealth project**

EDDL Components

- **Tensors**
  - N-dimensional memory structures used in a neural network model
  - Tensors have an associated buffer where data is stored
  - Tensors have a shape and are assigned to an specific device (CPU, GPU, FPGA)
  - Most frequent tensor operations implemented
- **Layers**
  - Layers of the neural network, each type has an associated class
  - Layers have pointers to tensors to store inputs, outputs, weights, bias, additional temporary buffers (gradients, ...)

EDDL Components

- **Computing Services**
  - Target device to run the training/inference process
  - Tensors allocated in the target device & accessed from its memory
  - Devices: CPU (Eigen), GPU (CUDA, CUDNN), FPGA (OpenCL)
  - Extension to COMPSs for distributed training
    - Node-level parallelism exploited by EDDL
    - Inter-node communication and synchronization exploited by COMPSs
- **Neural network specific**
  - Looses, metrics, regularizers, initializers, optimizers, …
- **API**
  - User level programming interface to abstract away all EDDL library
  - Documentation: https://deephealthproject.github.io/eddl/

```cpp
#include "eddl/apis/eddl.h"

using namespace eddl;

int main(int argc, char **argv) {

    download_mnist();   // Download mnist

    // Settings
    int epochs = 10;
    int batch_size = 200;
    int num_classes = 10;

    // Define network
    layer in = Input({784});
    layer l = in;  // Aux var
    l = LeakyReLu(Dense(l, 1024));
    l = LeakyReLu(Dense(l, 1024));
    l = LeakyReLu(Dense(l, 1024));
    layer out = Softmax(Dense(l, num_classes), -1);

    model net = Model({in}, {out});

    // Build model
    build(net,
          adam(0.001), // Optimizer
          {"softmax_cross_entropy"}, // Losses
          {"categorical_accuracy"}, // Metrics
          CS_CPU());

    summary(net);    // View model

    // Load dataset
    Tensor* x_train = Tensor::load("mnist_trX.bin");
    Tensor* y_train = Tensor::load("mnist_trY.bin");
    Tensor* x_test = Tensor::load("mnist_tsX.bin");
    Tensor* y_test = Tensor::load("mnist_tsY.bin");

    // Preprocessing
    x_train->div_(255.0f);
    x_test->div_(255.0f);

    // Train model
    fit(net, {x_train}, {y_train}, batch_size, epochs);

    // Evaluate
    evaluate(net, {x_test}, {y_test});

    // Release objects
    delete x_train;
    delete y_train;
    delete x_test;
    delete y_test;
    delete net;

    return EXIT_SUCCESS;
}
```

EDDL Example (C++)

eFlows4HPC

- **MNIST simplistic case (dense layers)**
- **Steps:**
  - Download dataset
  - Define network
  - Build model
    - Optimizer
    - Losses
    - Metrics
    - Computing service
  - Load dataset into tensors
  - Preprocessing
  - Fit the model
  - Evaluate the model
  - Delete memory resources

Coarse and Fine-grained Training

```
model net = Model({in}, {out});

// Build model
...

// Train model
fit(net, {x_train}, {y_train}, batch_size, epochs);
```

Coarse training simplifies the task and runs for a number of epochs using the train dataset

```
tshape s = x_train->getShape();
int num_batches = s[0]/batch_size;

for(i=0; i<epochs; i++) {
    reset_loss(net);
    for(j=0; j<num_batches; j++) {
        vector<int> indices = random_indices(batch_size, s[0]);
        train_batch(net, {x_train}, {y_train}, indices);
    }
}
```

Fine-grained training enables sophisticated training process, with a rich set of alternative methods

⊟ Fine-grained training

- random_indices
- next_batch
- train_batch
- eval_batch
- set_mode
- reset_loss
- forward
- zeroGrads
- backward
- update
- print_loss
- clamp
- compute_loss
- compute_metric
- newloss
- newmetric
- detach

Computing Service

```
build(net,
      sgd(0.01),              // Optimizer
      {"soft_cross_entropy"},   // Losses
      {"categorical_accuracy"}, // Metrics
      CS_CPU(4),                // CPU with 4 threads
);
```

```
build(imported_net,
      sgd(0.01),              // Optimizer
      {"soft_cross_entropy"},   // Losses
      {"categorical_accuracy"}, // Metrics
      CS_GPU({1}),              // one GPU
      false
);
```

```
build(imported_net,
      sgd(0.01),              // Optimizer
      {"soft_cross_entropy"},   // Losses
      {"categorical_accuracy"}, // Metrics
      CS_FPGA({1}),             // FPGA
);
```

```
build(imported_net,
      sgd(0.01f),                    // Optimizer
      {"soft_cross_entropy"},          // Losses
      {"categorical_accuracy"},        // Metrics
      CS_COMPSS("filename.cfg"),       // COMPSS config file
);
```

Computing can be moved to any set of devices by using the computing service method.

Functions exist to move computing between devices.

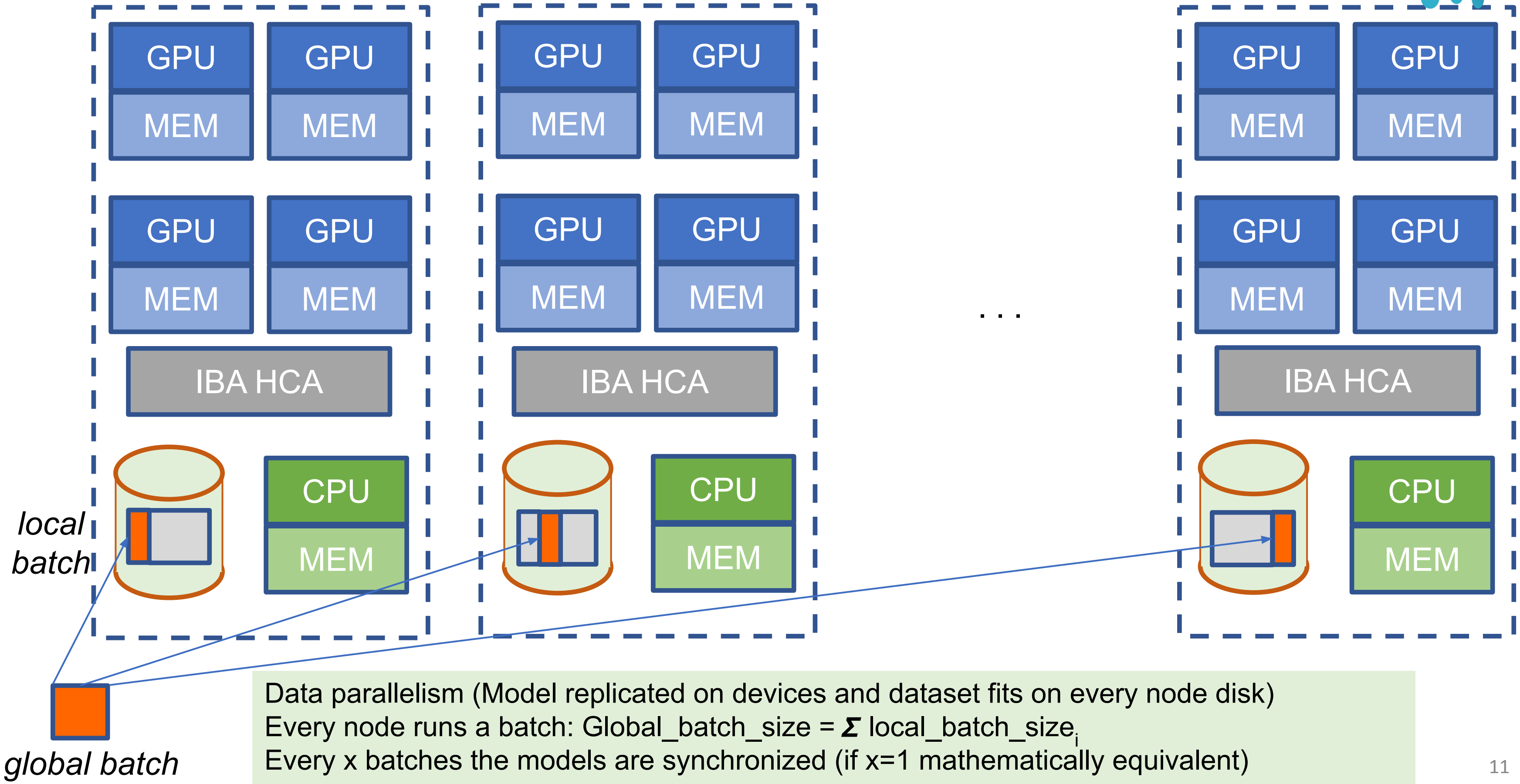
Distributed Training (MPI/NCCL)

eFlows4HPC

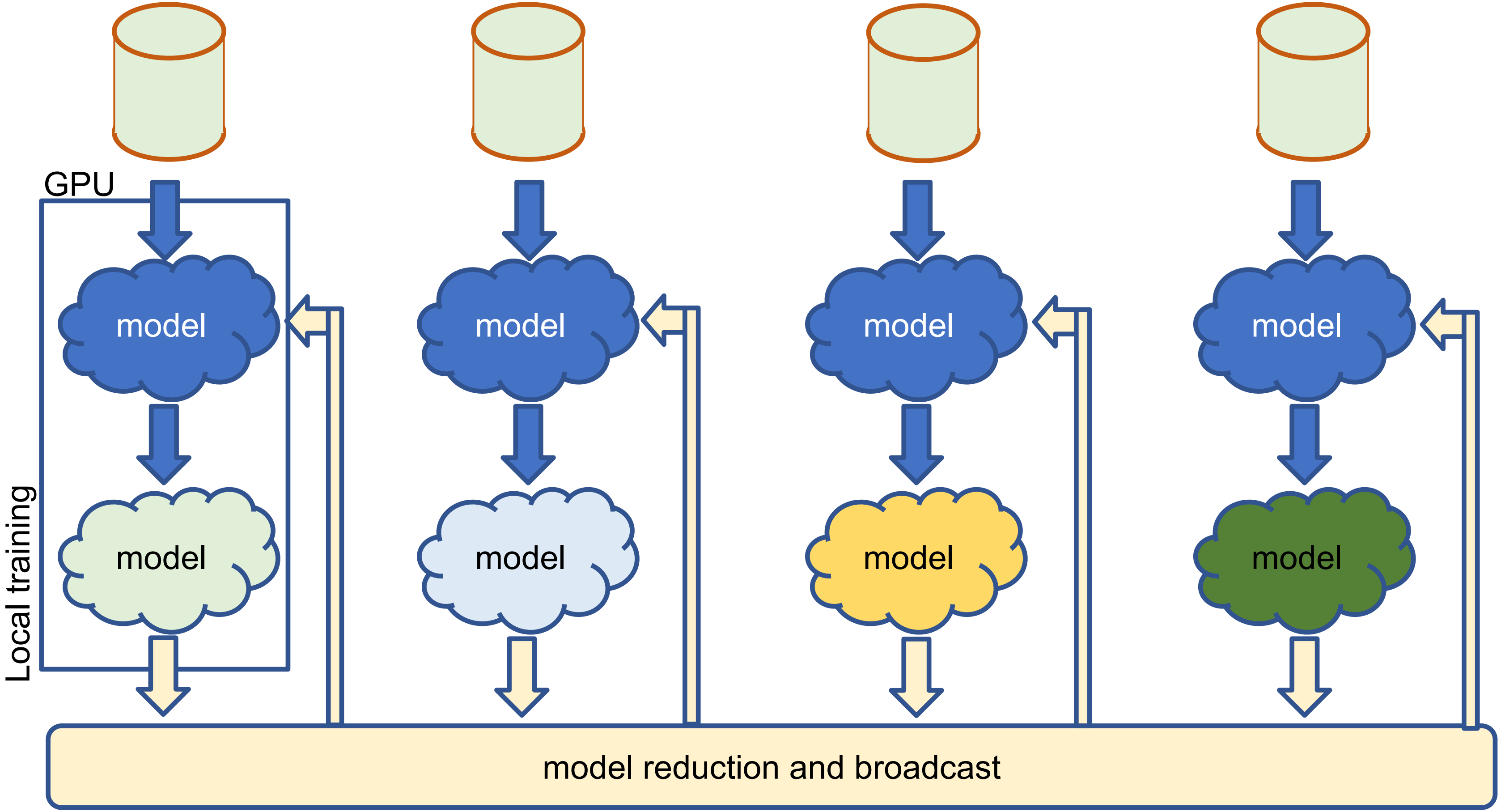
```
#include "eddl/apis/eddl.h"

using namespace eddl;

int main(int argc, char **argv) {

    int batch_size = 800;
    int epochs = 10;

    init_distributed("MPI"); // init_distributed("NCCL");
    id=get_id_distributed();    // Get MPI process id

    // Sync every batch, change every 1 epochs
    set_avg_method_distributed(LIMIT_OVERHEAD,8,0.1);

    download_mnist();   // Download mnist

    layer in = Input({1, 28, 28});
    l = Flatten(in);
    l = LeakyReLu(Dense(l, 1024));
    l = LeakyReLu(Dense(l, 1024));
    l = LeakyReLu(Dense(l, 10));
    layer out = Softmax(Dense(l, num_classes), -1);
    model net = Model({in},{out});

    // Build model
    build(net,
            adam(0.001),               // Optimizer
            {"softmax_cross_entropy"}, // Losses
            {"categorical_accuracy"},  // Metrics
            CS_CPU({}));

    Tensor* x_train = Tensor::load("mnist_trX.bin");    // Load dataset
    Tensor* y_train = Tensor::load("mnist_trY.bin");
    Tensor* x_test = Tensor::load("mnist_tsX.bin");
    Tensor* y_test = Tensor::load("mnist_tsY.bin");

    x_train->div_(255.0f);    // Preprocessing
    x_test->div_(255.0f);

    // Train model
    fit(net,{x_train},{y_train}, batch_size, epochs);

    // Evaluate
    evaluate(net,{x_test}, {y_test}); //evaluate_distr(net,{x_test}, {y_test});

    // Release objects, layers, ...
    delete x_train;
    delete y_train;
    delete x_test;
    delete y_test;
    delete net;

    // Finalize distributed training
    end_distributed();

    return EXIT_SUCCESS;
}
```

- **Trasparent distributed training process**
    - MPI and NCCL
    - Intranode and internode
    - Different synchronization policies
        - Fully synchronous
        - Synchronization every x batches
        - Dynamic (bounded communication overhead)
    - Almost linear scalability shown on Power9 cluster

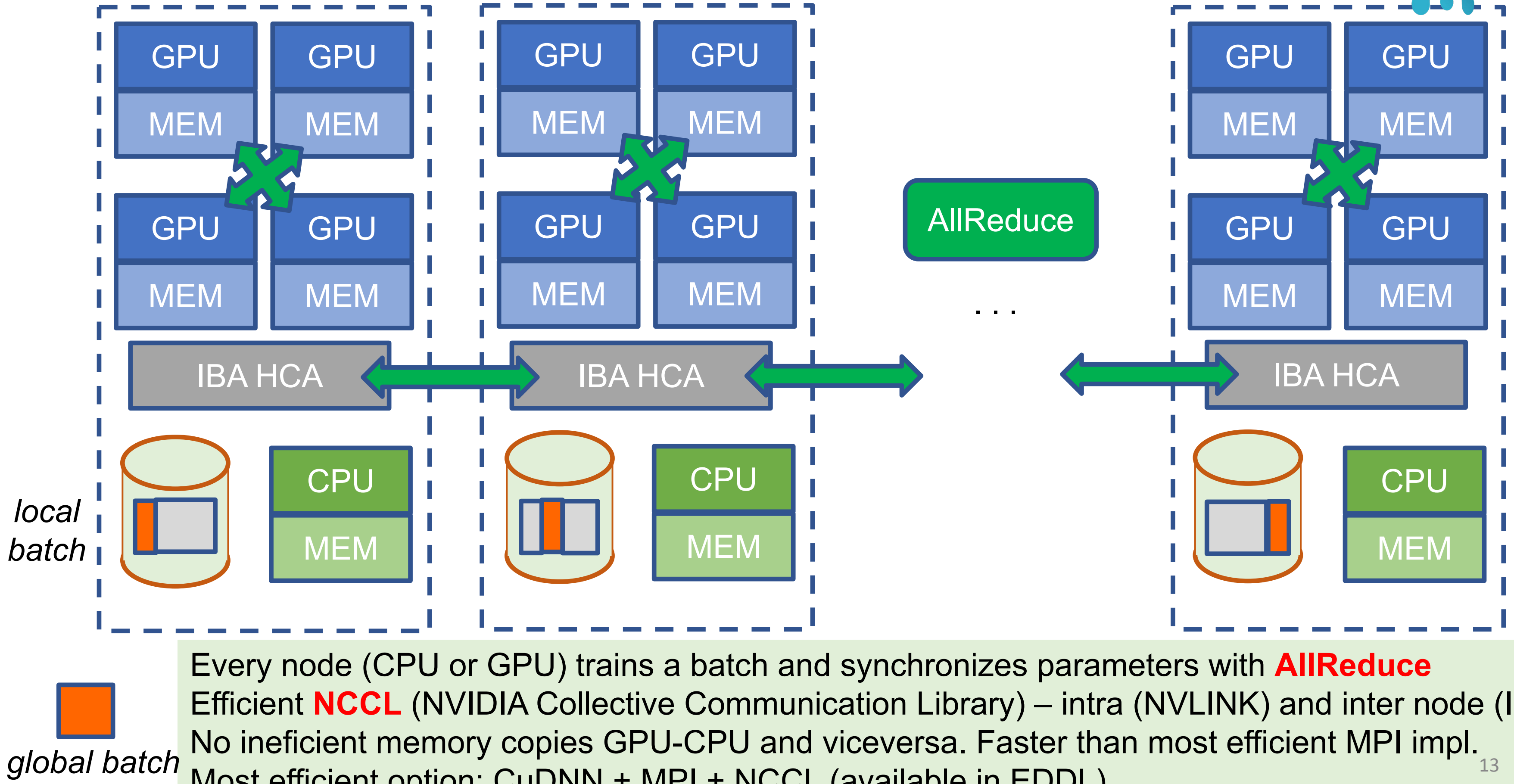
Distributed Training (native MPI support)



Data parallelism (Model replicated on devices and dataset fits on every node disk)
Every node runs a batch: Global_batch_size = Σ local_batch_size$_i$
Every x batches the models are synchronized (if x=1 mathematically equivalent)


GPU
Local training
model
model
model
model
model
model
model
model
model reduction and broadcast

Distributed Training (native MPI support)



Every node (CPU or GPU) trains a batch and synchronizes parameters with **AllReduce**
Efficient **NCCL** (NVIDIA Collective Communication Library) – intra (NVLINK) and inter node (IBA).
No ineficient memory copies GPU-CPU and viceversa. Faster than most efficient MPI impl.
Most efficient option: CuDNN + MPI + NCCL (available in EDDL)

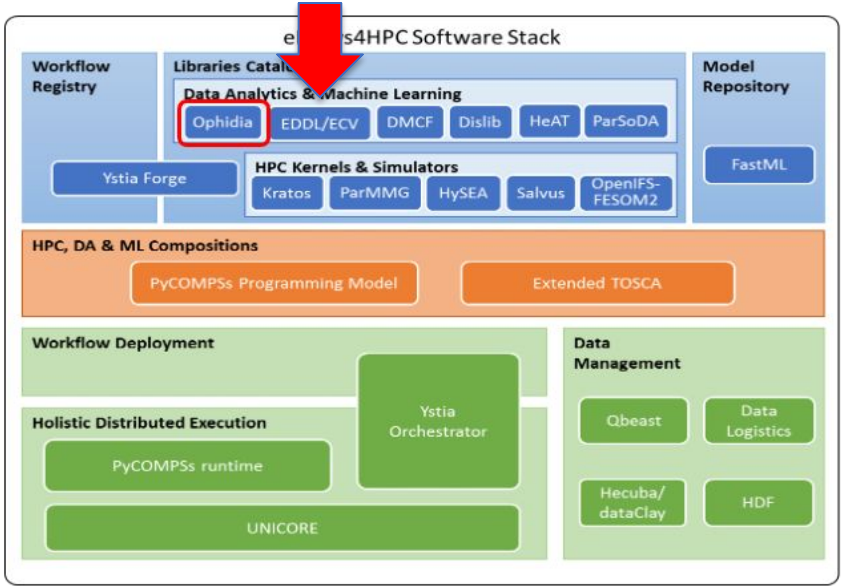
EDDL Compatibility (via ONNX)

- **Models can be saved and loaded in ONNX format**
  - Enables compatibility with other tools
- **A binary format (EDDL proprietary) exists**

```
save_net_to_onnx_file(net, "my_model.onnx");
```

```
Net* net = import_net_from_onnx_file("my_model.onnx");
```

EDDL in eFlows4HPC



EDDL is one of the ML tools in the **eFlows4HPC Software stack.**

Supports training and inference of neural network models needed in:

- **Pillar I Reduced Order Models:** Autoencoders
- **Pillar II Earth System Model workflow:** Cyclone tracking
- **Pillar III Tsunami workflow and EarthQuake workflow**

Motivations:

- Need of neural network models

Summary

- EDDL provides a complete software stack to run sequential and distributed neural network training processes, as well as inference processes (including FPGAs)

- PyEDDL Python wrapper enable python development abstracting from the infrastructure complexity

- Integration of EDDL with PyCOMPSs through Computing Service method

- Initial implementation of scientific use cases in the eFlows4HPC project with EDDL successful

Questions

eFlows4HPC

Enabling dynamic and Intelligent workflows
in the future EuroHPC ecosystem

www.eFlows4HPC.eu

@eFlows4HPC   eFlows4HPC Project

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

Part 2: Ophidia for HPDA in Project Pillars

Donatello Elia
Euro-mediterranean Center on Climate Change (CMCC)

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union’s Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

Ophidia HPDA Framework

eFlows4HPC

Ophidia (http://ophidia.cmcc.it) is a CMCC Foundation research project addressing data challenges for eScience with a focus on the climate domain

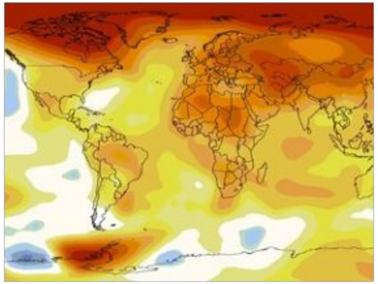
- A **HPDA framework** for multi-dimensional scientific data joining HPC paradigms with scientific data analytics approaches
- **In-memory** and **server-side** data analysis exploiting parallel computing techniques
- Multi-dimensional, array-based, storage model and partitioning schema for scientific data leveraging the **datacube** abstraction
- End-to-end mechanisms to support **interactive analysis**, **complex experiments** and **large workflows** on scientific data

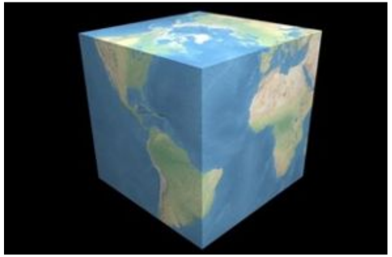
Ophidia

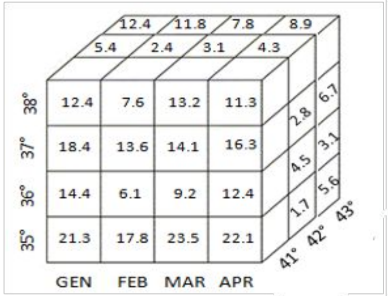
cmcc
Centro Euro-Mediterraneo
sui Cambiamenti Climatici

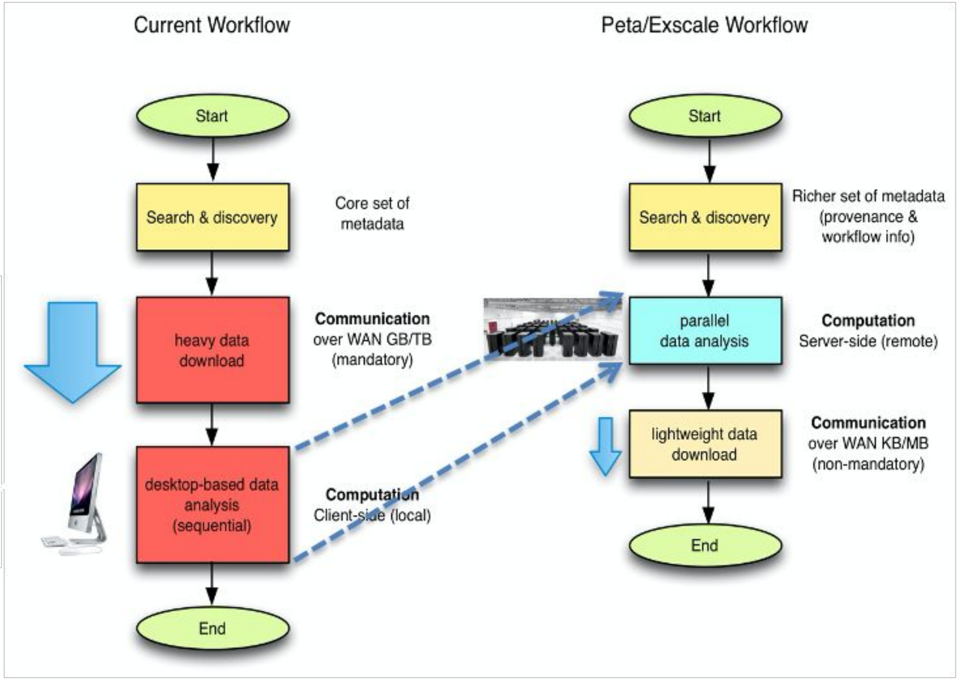
A paradigm shift: from Desktop to Server analysis

Volume, variety, velocity are key challenges for big data in general and for climate sciences in particular. Client-side, sequential and disk-based workflows are three limiting factors for the current scientific data analysis tools.







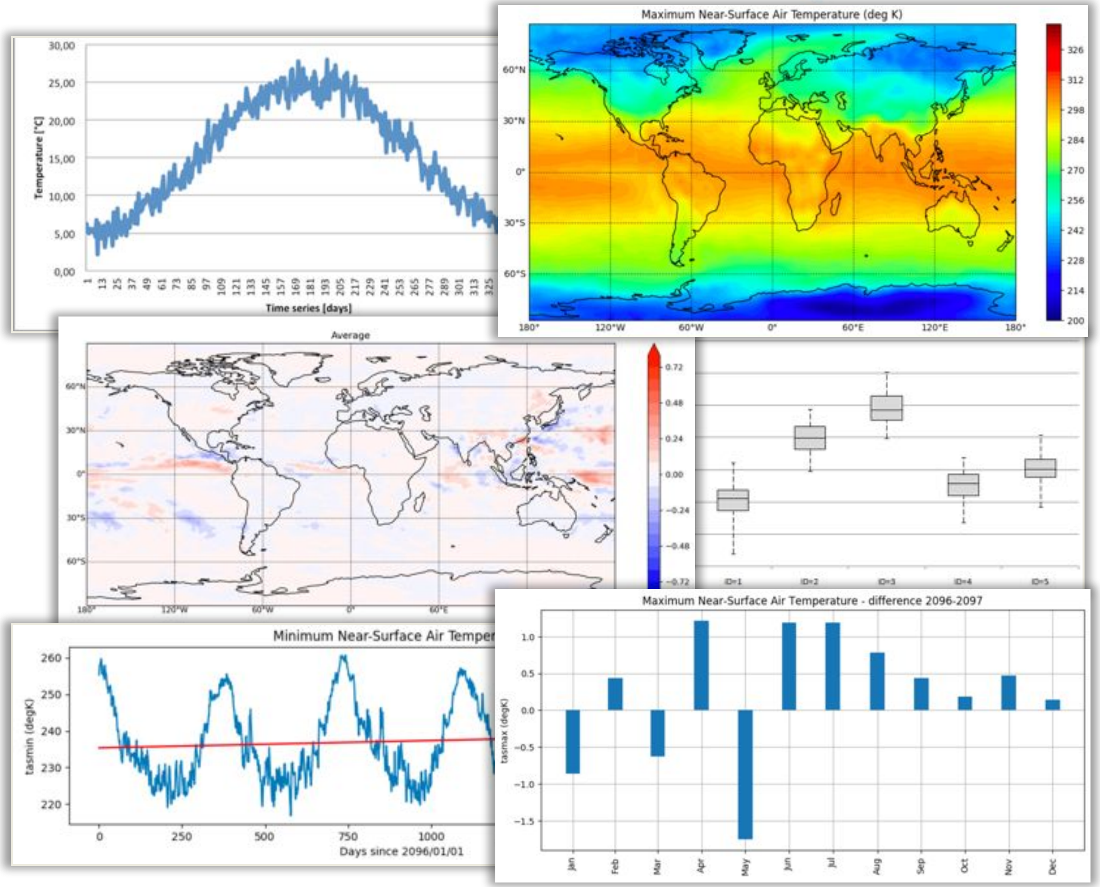


S. Fiore, A. D'Anca, C. Palazzo, I. Foster, D. N. Williams, G. Aloisio, "Ophidia: toward bigdata analytics for eScience", ICCS2013 Conference, Procedia Elsevier, 2013

Use cases/applications supported by Ophidia

eFlows4HPC

- *Time series analysis*
- *Data subsetting*
- *Model intercomparison*
- *Multi-model means*
- *Massive data reduction*
- *Data transformation*
- *Parameter sweep experiments*
- *Ensemble analysis*
- *Data analytics workflows*
- *Maps generation*
- *Data provenance*



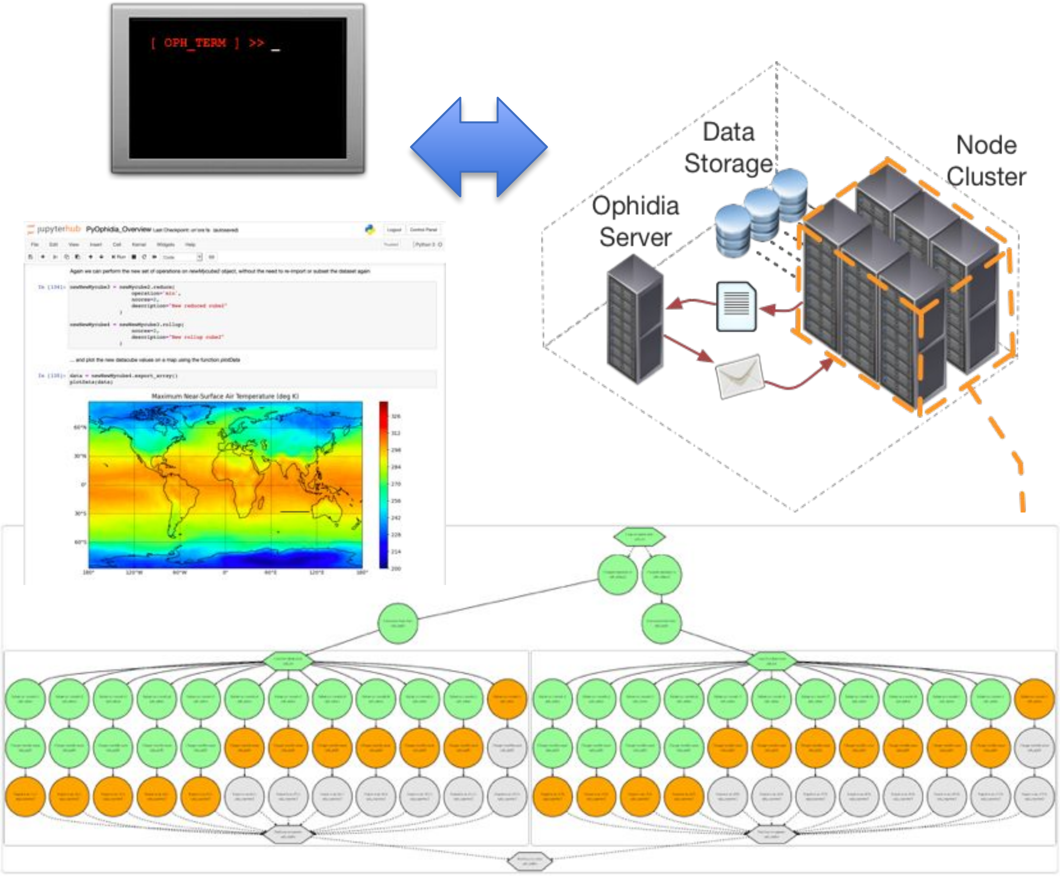
Ophidia Operators

About 50 operators for data (cube) and metadata processing

CLASS	PROCESSING TYPE	OPERATOR(S)
I/O	Parallel	OPH_IMPORTNC, OPH_EXPORTNC, OPH_CONCATNC, OPH_RANDUCUBE
Time series processing	Parallel	OPH_APPLY
Datacube reduction	Parallel	OPH_REDUCE, OPH_REDUCE2, OPH_AGGREGATE
Datacube subsetting	Parallel	OPH_SUBSET
Datacube combination	Parallel	OPH_INTERCUBE, OPH_MERGECUBES
Datacube structure manipulation	Parallel	OPH_SPLIT, OPH_MERGE, OPH_ROLLUP, OPH_DRILLDOWN, OPH_PERMUTE
Datacube/file system management	Sequential	OPH_DELETE, OPH_FOLDER, OPH_FS
Metadata management	Sequential	OPH_METADATA, OPH_CUBEIO, OPH_CUBESCHEMA
Datacube exploration	Sequential	OPH_EXPLORECUBE, OPH_EXPLORENC

Ophidia operators documentation: http://ophidia.cmcc.it/documentation/users/operators/index.html

Server-side paradigm and execution modes



Oph_Term: a terminal-like commands interpreter serving as a client for the Ophidia framework

PyOphidia: a Python interface for datacube management & analytics with Ophidia

Multiple execution modes:

- *Interactive analysis (e.g. notebooks)*
- *Python applications*
- *Workflows of operators*
- *Async/sync execution*

PyOphidia: programmatic support for data science

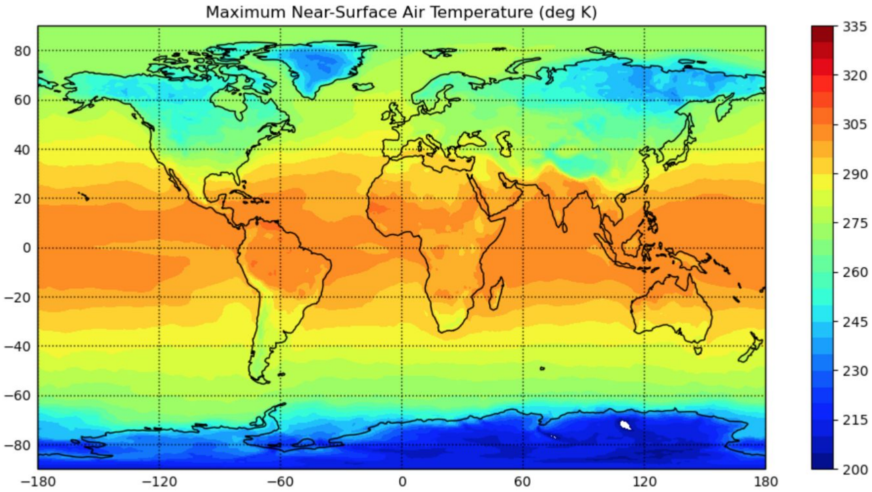
PyOphidia is a Python module to interact with the Ophidia framework

High-level and easy-to-use bindings for the HPDA framework:

- *APIs to manage deployment, data distribution and computation parallelism*
- *Management of (remote) data objects in the form of datacubes*
- *Easy exploitation from Jupyter*
- *Integration with other Python modules (i.e, maps)*
- *Conversion methods to Xarray and Pandas*

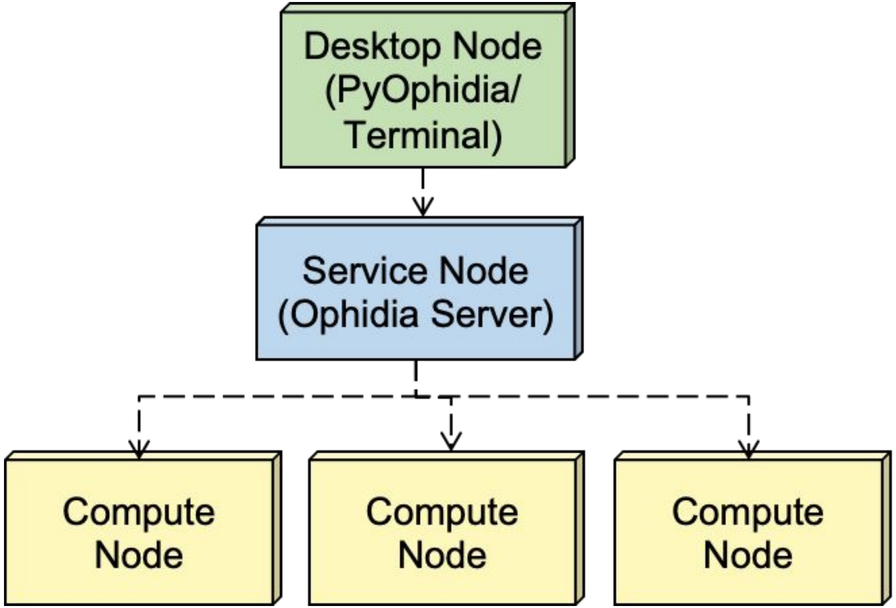
```
from PyOphidia import cube, client
cube.Cube.setclient(read_env=True)

mycube =
cube.Cube.importnc(src_path='/public/data/ecas_training
/file.nc', measure='tos', imp_dim='time',
import_metadata='yes', ncores=5)
mycube2 = mycube.reduce(operation='max',ncores=5)
mycube3 = mycube2.rollup(ncores=5)
data = mycube3.export_array()

mycube3.exportnc2(output_path='/home/test',
export_metadata='yes')
```

```
[29]: data = newNewMycube3.export_array()
      plotData(data)
```

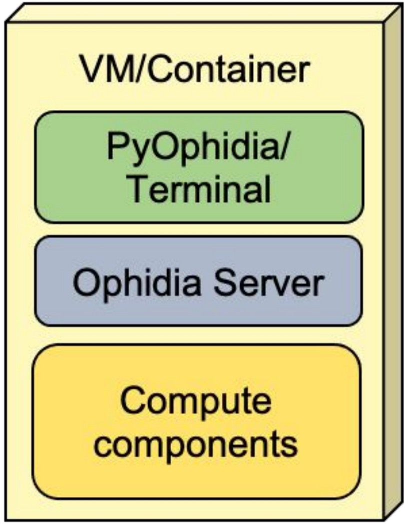


Support for different deployments

Multi-node setup



Single node setup

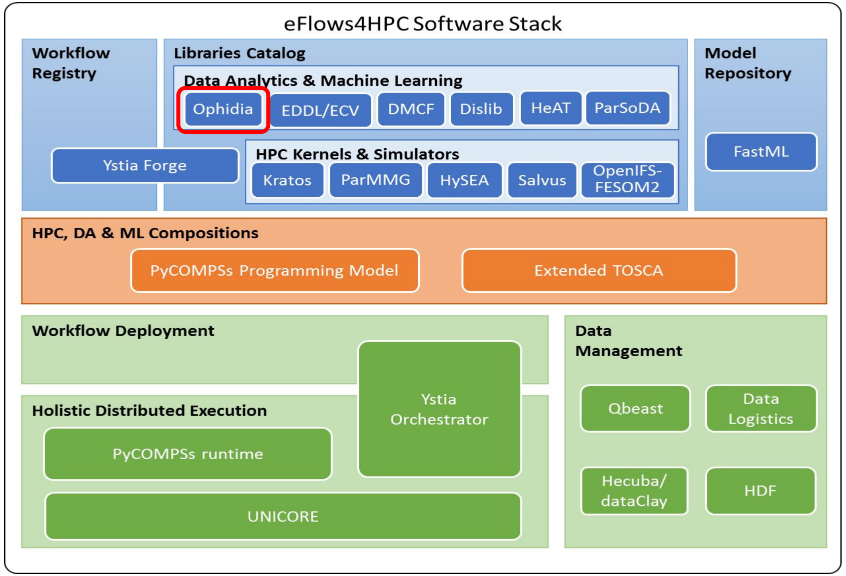


Ophidia architecture allows for flexible deployment targeting different scenarios:

- Distributed and scalable processing on top of **HPC** and **Cloud** infrastructures
- All-in-one local setup for training, testing and small-scale parallel analysis

Ophidia in eFlows4HPC

eFlows4HPC



Ophidia is one of the data analytics frameworks in the **eFlows4HPC Software stack.**

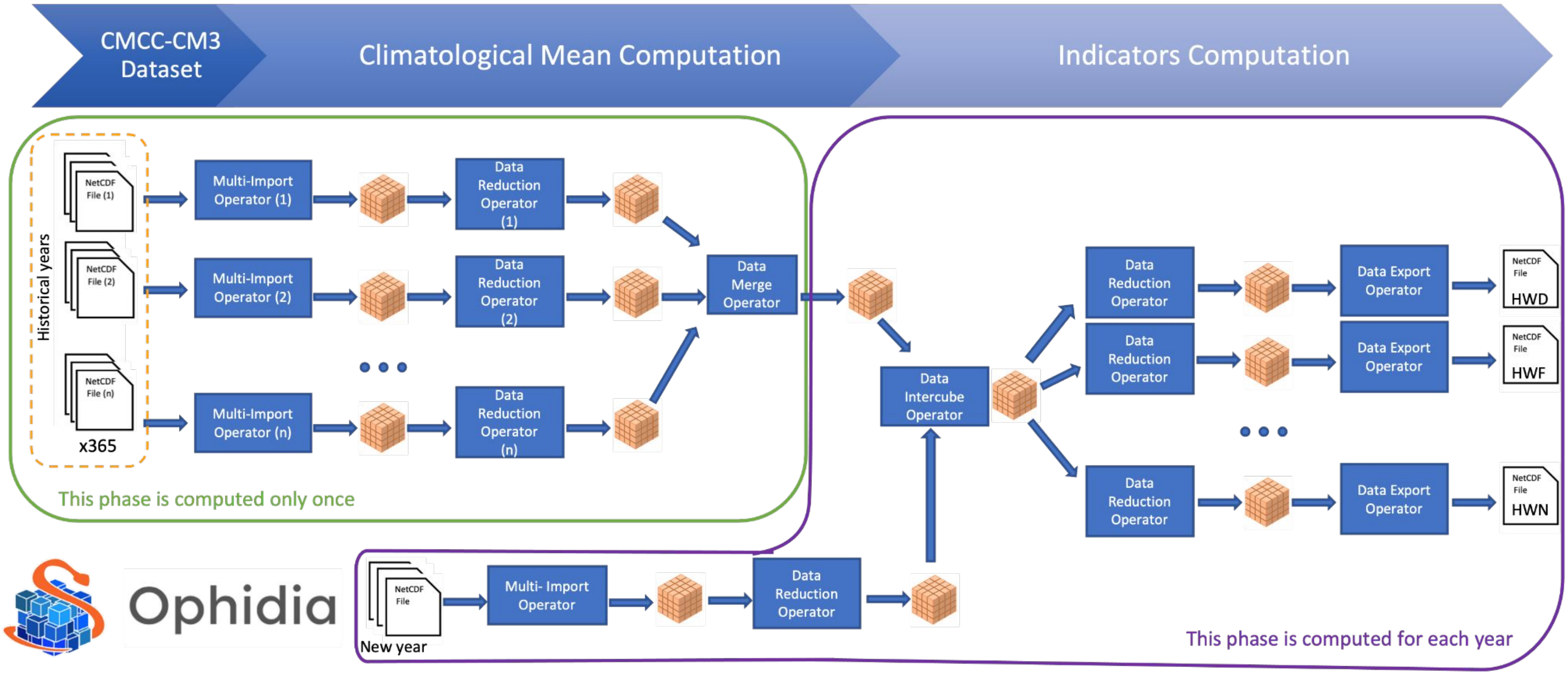
Supports pre-processing and computation of climate/environmental indices on simulation data for two project applications:

- **Pillar II Earth System Model workflow:** extreme event indicators (e.g., Heat Waves Number)
- **Pillar III Tsunami workflow:** tsunami metrics (e.g., max, min, peak-to-trough)

Motivations:

- Availability of parallel operators
- Native support for I/O on NetCDF data
- In-memory data management

Pillar II: extreme events indicators workflow

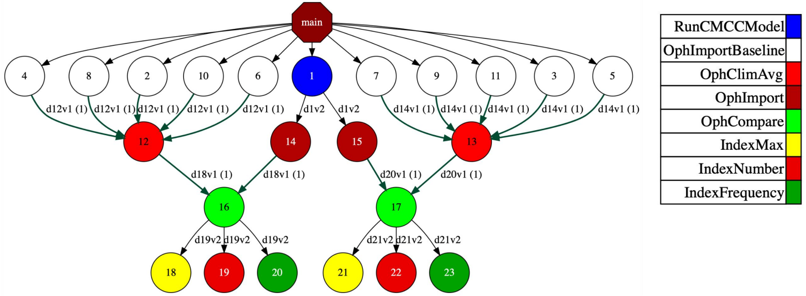


Pillar II: extreme events indicators

*Computation of extreme events indicators on each year produced by the model (**Heat Waves Number**, **Cold Waves Frequency**, etc.)*

Extensions to Ophidia in eFlows4HPC to:

- *speed-up import of **multiple files** into a **single datacube***
- *In-memory function for **climatological mean computation***



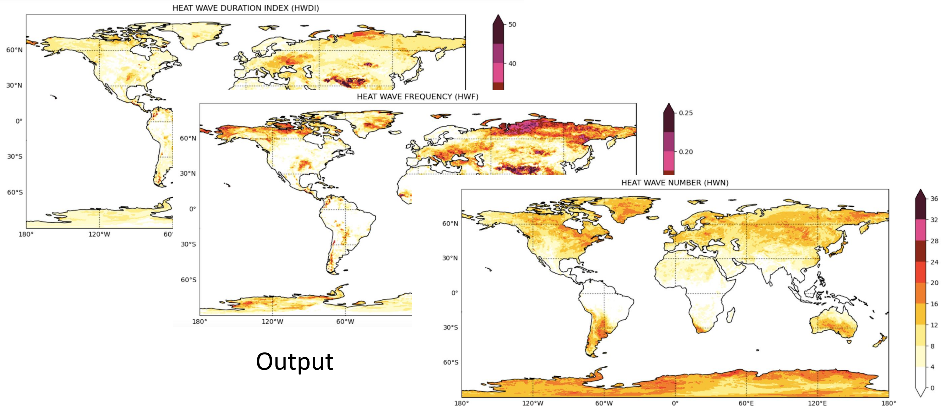
Baseline computation:

```
@task(returns=object)
def OphidiaImport(file, measure, op):
    src_path='2000_cam6-nemo4_025deg_tc/atm/hist/2000_cam6-nemo4_025deg_tc.cam.h1.' + file + '-*-*-00000.nc'
    Year = cube.Cube.importncs(src_path= src_path,
    measure=measure,
    imp_dim='time',
    description='6-Hours Temps'
    )
    movingAvg = Year.apply(
    query="oph_shift(oph_moving_avg(oph_reduce2(measure," + op + ",4), 5, 'OPH_SMA'), -2, 0)",nthreads=2)
    return movingAvg

@task(returns=object, movingavgcubes=COLLECTION_IN)
def OphidiaClimatologicalAvg(movingavgcubes, filename):
    baseline=cube.Cube.intercube2(cubes=movingavgcubes)
    baseline.exportnc2(output_path="/work/asc/ss18121/CMCC-CM3/HeatWaveNotebooks",output_name=filename)
    return baseline
```

Indicators computation:

```
@task(returns=object, cubes=COLLECTION_IN)
def OphidiaCompare(cubes, firstvalues, mask):
    #Computation of the difference between a year and the baseline;
    Diff = cubes[0].intercube(cube2=cubes[1].pid, operation='sub')
    Duration = Diff.apply(
    query="oph_predicate(oph_sequence(oph_predicate("+firstvalues+",'x-5',"+mask+",'1','0')),'x-5','>0','x','0')")
    return Duration

@task(returns=object)
def IndexDurationNumber(duration, filename):
    #Computation of the number of durations in a year (HWN or CWN)
    DurationMask = duration.apply(query="oph_predicate('OPH_INT','OPH_INT',measure,'x','>0','1','0')")
    DurationCount = DurationMask.reduce(operation='sum', ncores=1, description="Number of durations cube")
    return DurationCount
```

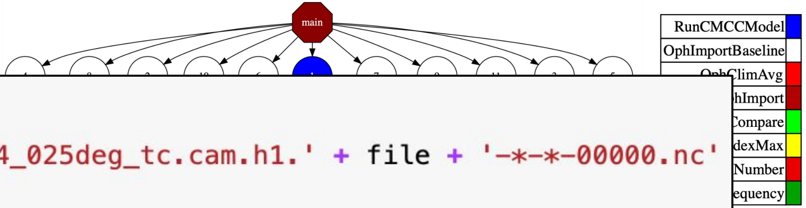
PyCOMPSs is used to perform the Ophidia pipelines concurrently on different input files and orchestrate the execution of the various operators.



Output

Pillar II: extreme events indicators

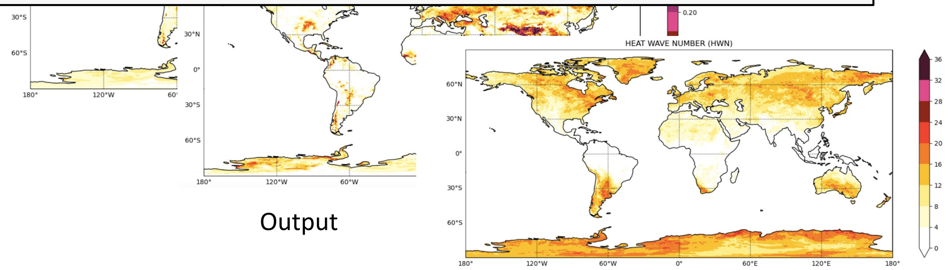
*Computation of extreme events indicators on each year produced by the model (**Heat Waves Number**, **Cold Waves Frequency**, etc.)*




```
@task(returns=object)
def OphidiaImport(file, measure, op):
    src_path='2000_cam6-nemo4_025deg_tc/atm/hist/2000_cam6-nemo4_025deg_tc.cam.h1.' + file + '-*-*-00000.nc'
    Year = cube.Cube.importncs(src_path= src_path,
    measure=measure,
    imp_dim='time',
    description='6-Hours Temps'
    )
    movingAvg = Year.apply(
    query="oph_shift(oph_moving_avg(oph_reduce2(measure," + op + ",4), 5, 'OPH_SMA'), -2, 0)",nthreads=2)
    return movingAvg

@task(returns=object, movingavgcubes=COLLECTION_IN)
def OphidiaClimatologicalAvg(movingavgcubes, filename):
    baseline=cube.Cube.intercube2(cubes=movingavgcubes)
    baseline.exportnc2(output_path="/work/asc/ss18121/CMCC-CM3/HeatWaveNotebooks",output_name=filename)
    return baseline
```

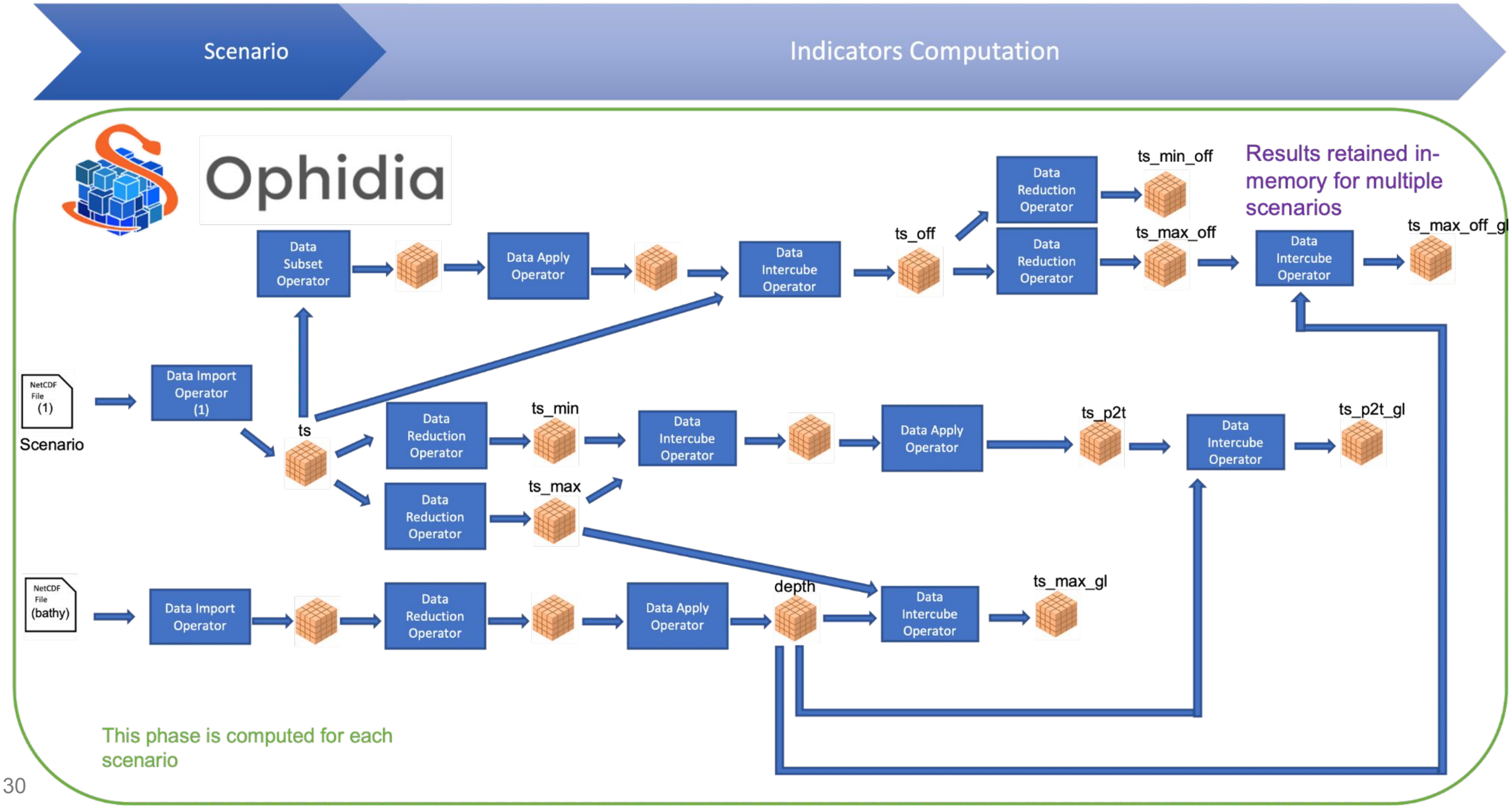
Indicators computation:

```
@task(returns=object, cubes=COLLECTION_IN)
def OphidiaCompare(cubes, firstvalues, mask):
    #Computation of the difference between a year and the baseline;
    Diff = cubes[0].intercube(cube2=cubes[1].pid, operation='sub')
    Duration = Diff.apply(
    query="oph_predicate(oph_sequence(oph_predicate("+firstvalues+",'x-5',"+mask+",'1','0')),'x-5','>0','x','0')")
    return Duration

@task(returns=object)
def IndexDurationNumber(duration, filename):
    #Computation of the number of durations in a year (HWN or CWN)
    DurationMask = duration.apply(query="oph_predicate('OPH_INT','OPH_INT',measure,'x','>0','1','0')")
    DurationCount = DurationMask.reduce(operation='sum', ncores=1, description="Number of durations cube")
    return DurationCount
```



Output

Pillar III: operations on tsunami data workflow

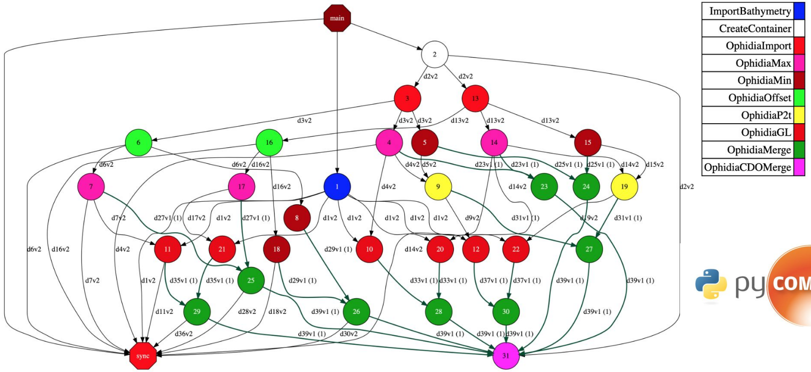


Pillar III: indicators computation

Computes for each of the time series from the tsunami simulations (**wave amplitude** variable): ***max***, ***min***, ***peak-to-trough***, ***green's amplification***, *etc.*

Extensions to Ophidia in eFlows4HPC to:

- *Better support I/O and management of **non-spatial oriented** data*



Indicators computation:

```python
#This task computes the ts_max
@task(returns=str)
def OphidiaMax(ts):
    cube.Cube.setclient(read_env=True, project="0459")
    ts_max = ts.reduce(operation="max")
    return ts_max.pid

#This task computes the ts_min
@task(returns=str)
def OphidiaMin(ts):
    cube.Cube.setclient(read_env=True, project="0459")
    ts_min = ts.reduce(operation="min")
    return ts_min.pid

#This task computes the ts_off
@task(returns=object)
def OphidiaOffset(ts):
    cube.Cube.setclient(read_env=True, project="0459")
    # Computation of ts_offset
    firstRow=ts.subset(subset_dims="time",subset_filter="1",subset_type="index")
    ts0 = firstRow.apply(query="oph_extend('OPH_FLOAT','OPH_FLOAT',measure,961)")
    ts_off = ts.intercube(cube2=ts0.pid, operation='sub')
    return ts_off
```

```
netcdf Step2_BS_hmax {
dimensions:
scenarios = 864 ;
grid_npoints = 1107 ;
variables:
double scenarios(scenarios) ;
double grid_npoints(grid_npoints) ;
float ts_max(grid_npoints, scenarios) ;
ts_max:long_name = "Wave amplitude" ;
float ts_min(grid_npoints, scenarios) ;
ts_min:long_name = "Wave amplitude" ;
float ts_max_off(grid_npoints, scenarios) ;
float ts_min_off(grid_npoints, scenarios) ;
float ts_p2t(grid_npoints, scenarios) ;
float ts_max_gl(grid_npoints, scenarios) ;
float ts_max_off_gl(grid_npoints, scenarios) ;
float ts_p2t_gl(grid_npoints, scenarios) ;
```

Summary

- Ophidia provides a complete software stack to run parallel, server-side in-memory analysis
- PyOphidia Python module represents a high-level interface to Ophidia abstracting from the infrastructure complexity
- Integration of Ophidia with PyCOMPSs through PyOphidia to support two eFlows4HPC pillar applications
- Initial implementation of scientific use cases in the eFlows4HPC project with PyOphidia/PyCOMPSs successful
- Full integration of Ophidia in the project software stack in progress

Questions

eFlows4HPC

Enabling dynamic and Intelligent workflows in the future EuroHPC ecosystem

www.eFlows4HPC.eu

@eFlows4HPC eFlows4HPC Project

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

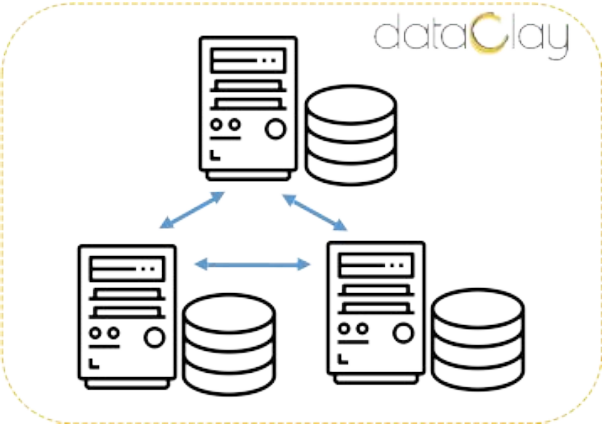
Part 3: dataClay, locality and enhanced iteration

Alex Barcelo

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

dataClay in a nutshell

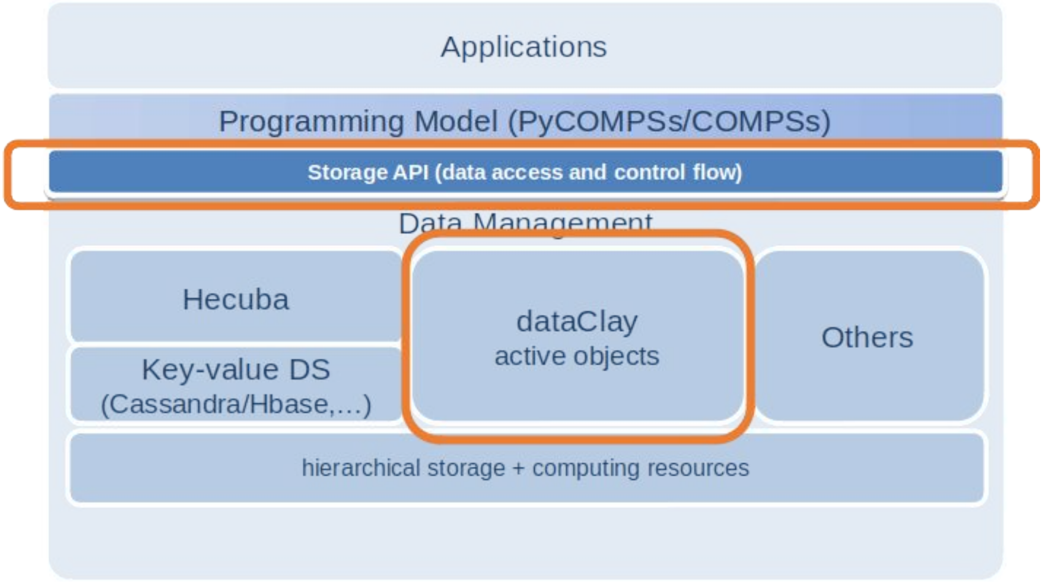
- Distributed *active object store* for HPC and big data applications
- A single data model to manage transparently:
  - Persistent and volatile data
  - Local and remote data
- Inherently exploits data locality
  - Objects = data + methods
  - Reduces data movements
- Backends keep objects already instantiated in memory
  - Objects ready to be used
  - Avoids transformations and access to disk



BSC's HPC software stack

Goal: Integrate persistent objects as naturally as possible with

- The programming language
- The programming model



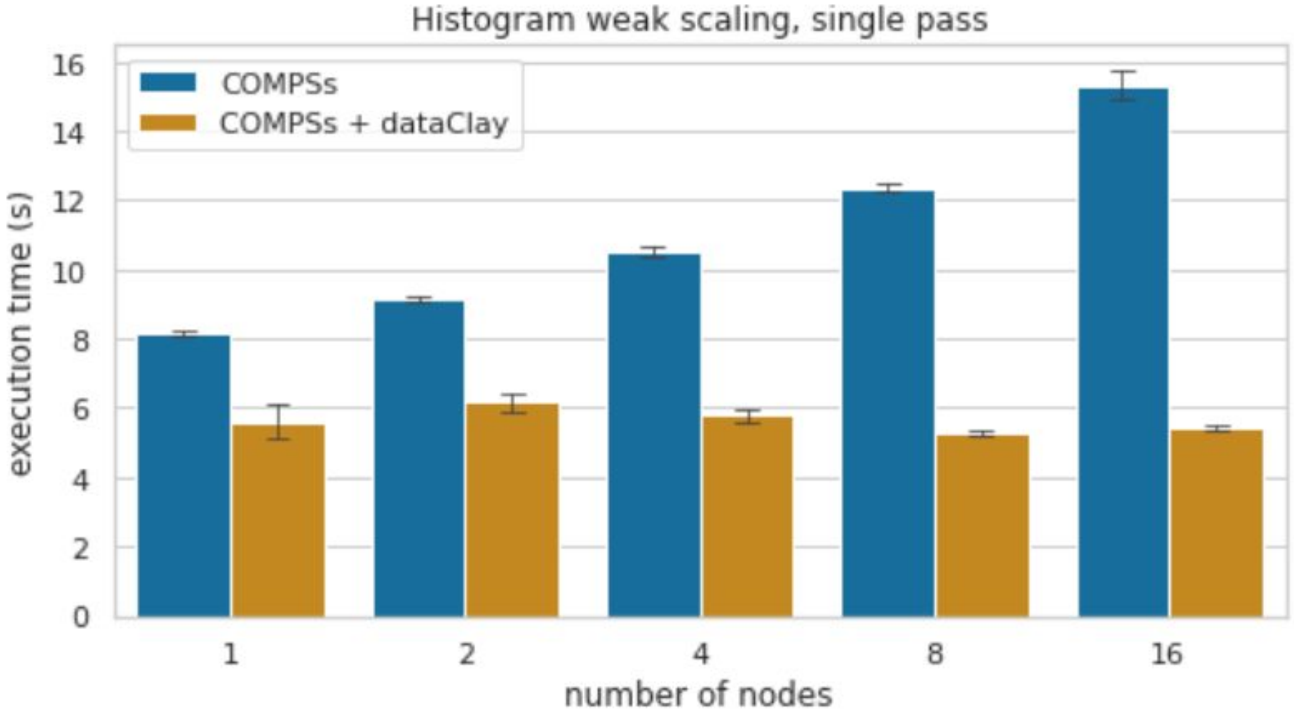
Active Methods - Developer POV

Move the code from a function into a method (within a dataClay class)

```
@task(returns=object)
def partial_histogram(fragment, n_bins, n_dimensions):
    values, _ = np.histogramdd(fragment, n_bins, [(0, 1)] * n_dimensions)
    return values
```

```
class PersistentBlock(DataClayObject):
    @task(target_direction=IN, returns=object)
    @dclayMethod(n_bins="int", n_dimensions="int", return_="numpy.ndarray")
    def partial_histogram(self, n_bins, n_dimensions):
        values, _ = np.histogramdd(self.block_data, n_bins, [(0, 1)] * n_dimensions)
        return values
```

Active Methods - Performance POV



DISCLAIMER: Blaming GPFS is always a safe course of action!

Locality -- is it always the solution?

It depends

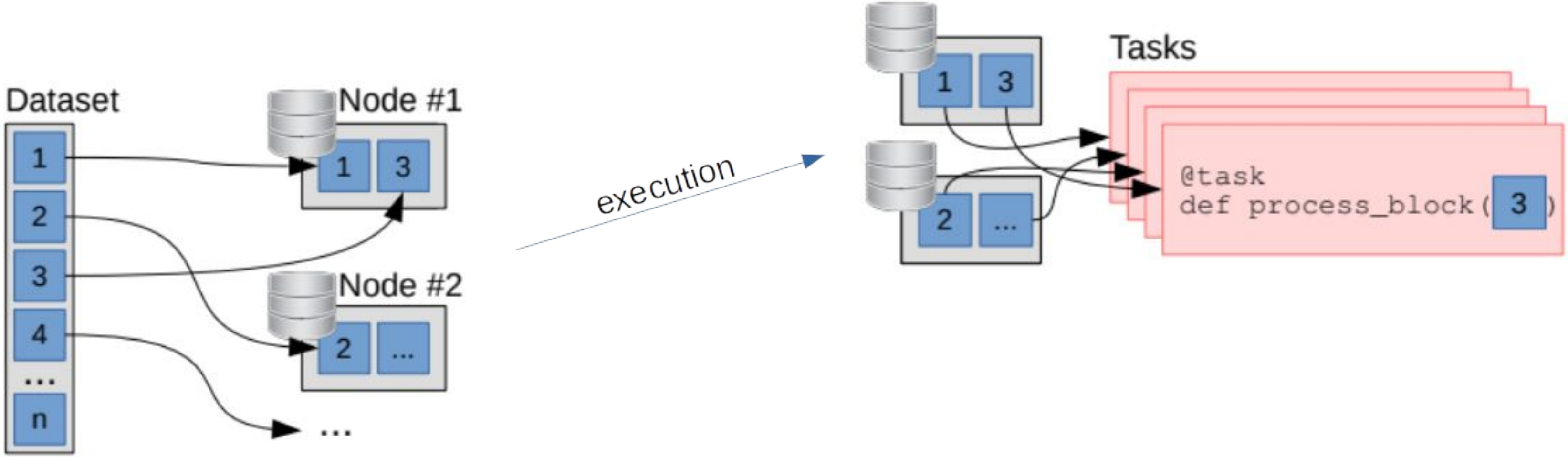
- **When tasks are memory intensive, it yields benefits**
- **When tasks are heavily compute bound, it doesn't matter**

Or, said differently:

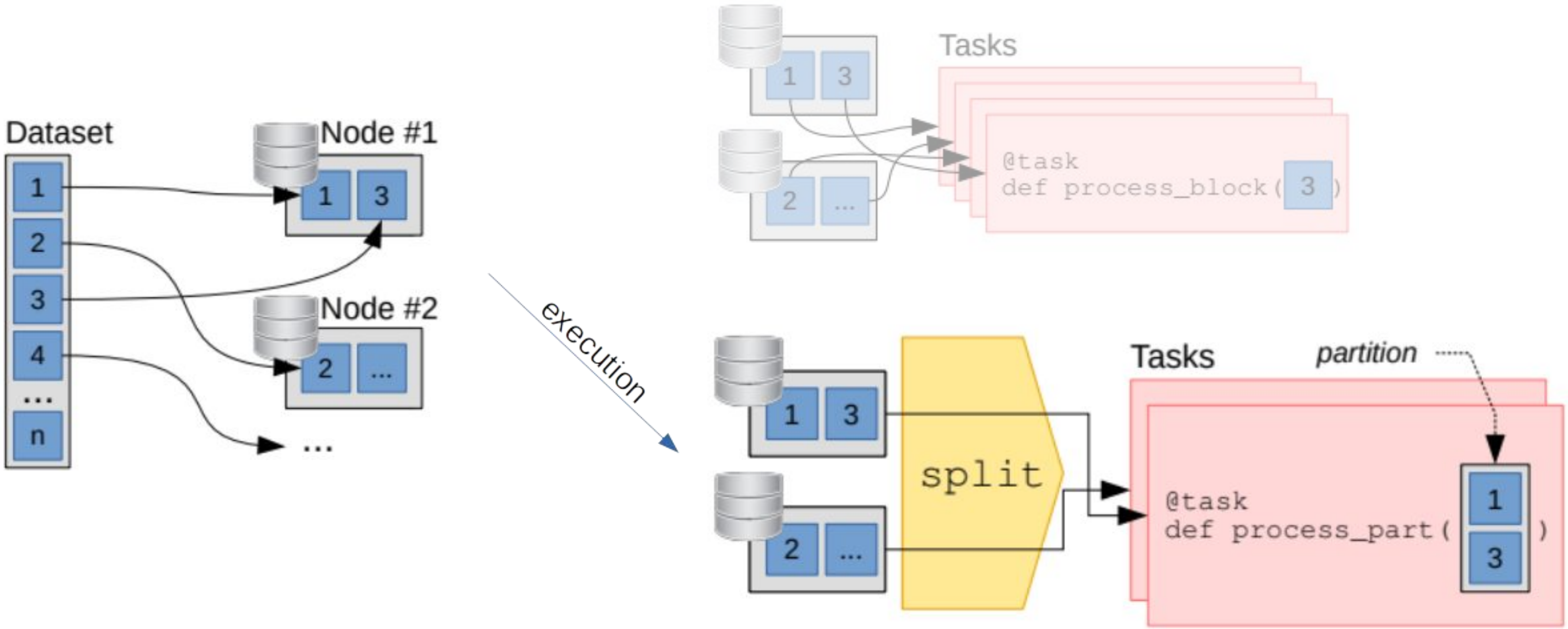
- If transfers and deserialization are your bottleneck, then add locality!
- If >90% of your time is pure computation, focus on the algorithm or hardware

Enhanced iteration with dataClay: *split*

Fundamentals (I)



Fundamentals (II)



split usage

Original with active dataClay methods:

```
partials = list()

for row in experiment._blocks:
    fragment = row[0]
    partial = fragment.partial_histogram(n_bins)
    partials.append(partial)
result = sum_partials(partials)
```

Original with no active methods:

```
partials = list()

for fragment in experiment._blocks:
    partial = partial_histogram(fragment, n_bins)
    partials.append(partial)
result = sum_partials(partials)
```

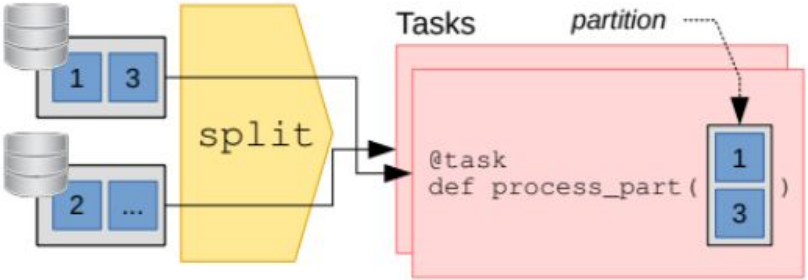
split version:

```
partials = list()
for partition in split(experiment):
    p = compute_partition(partition, n_bins)
    partials.append(p)
result = sum_partials(partials)
```

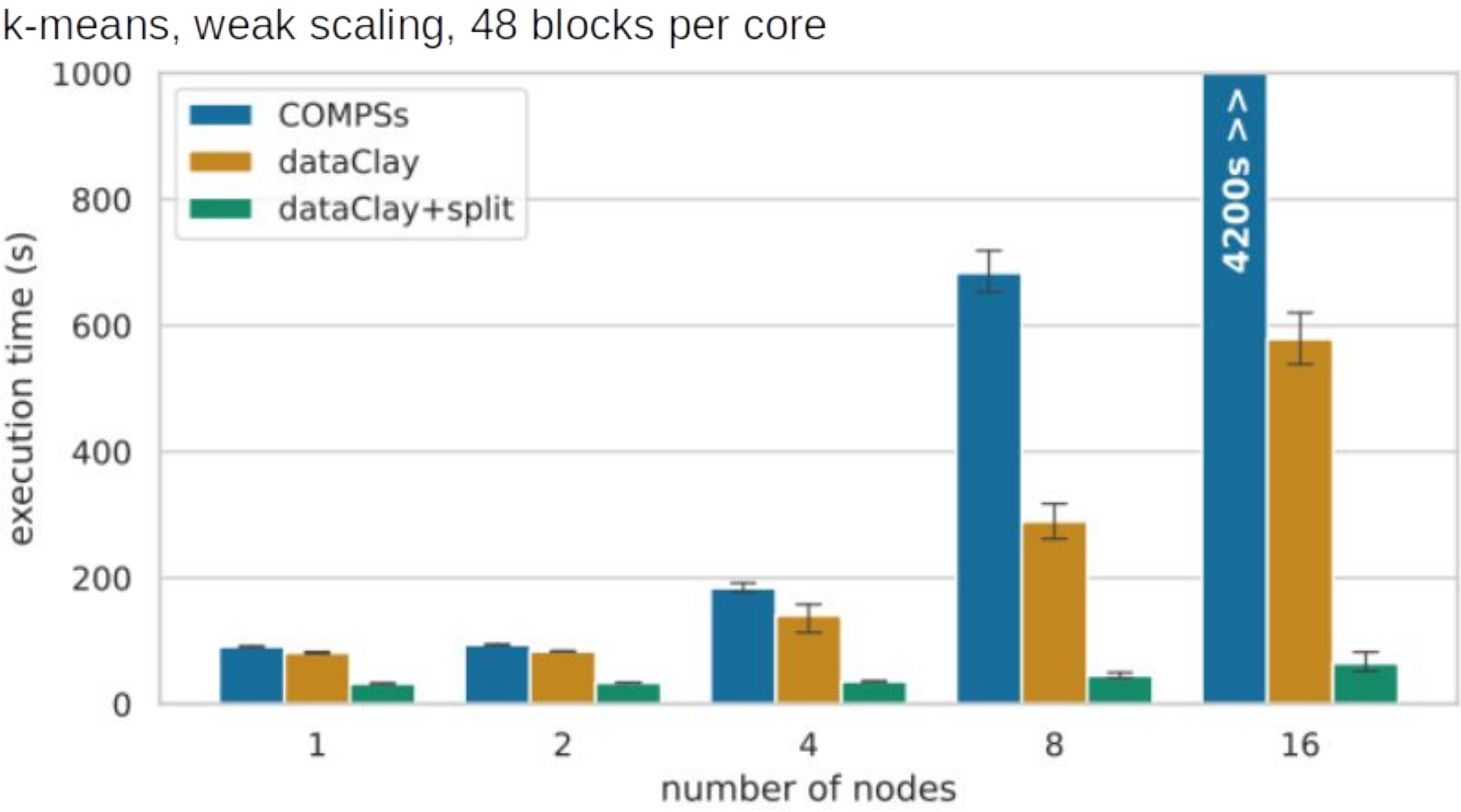
```
@task
def compute_partition(partition, n):
    subresults = list()
    for fragment in partition:
        partial = fragment.partial_histogram(n)
        subresults.append(partial)
    return sum_partials(subresults)
```

Advantages

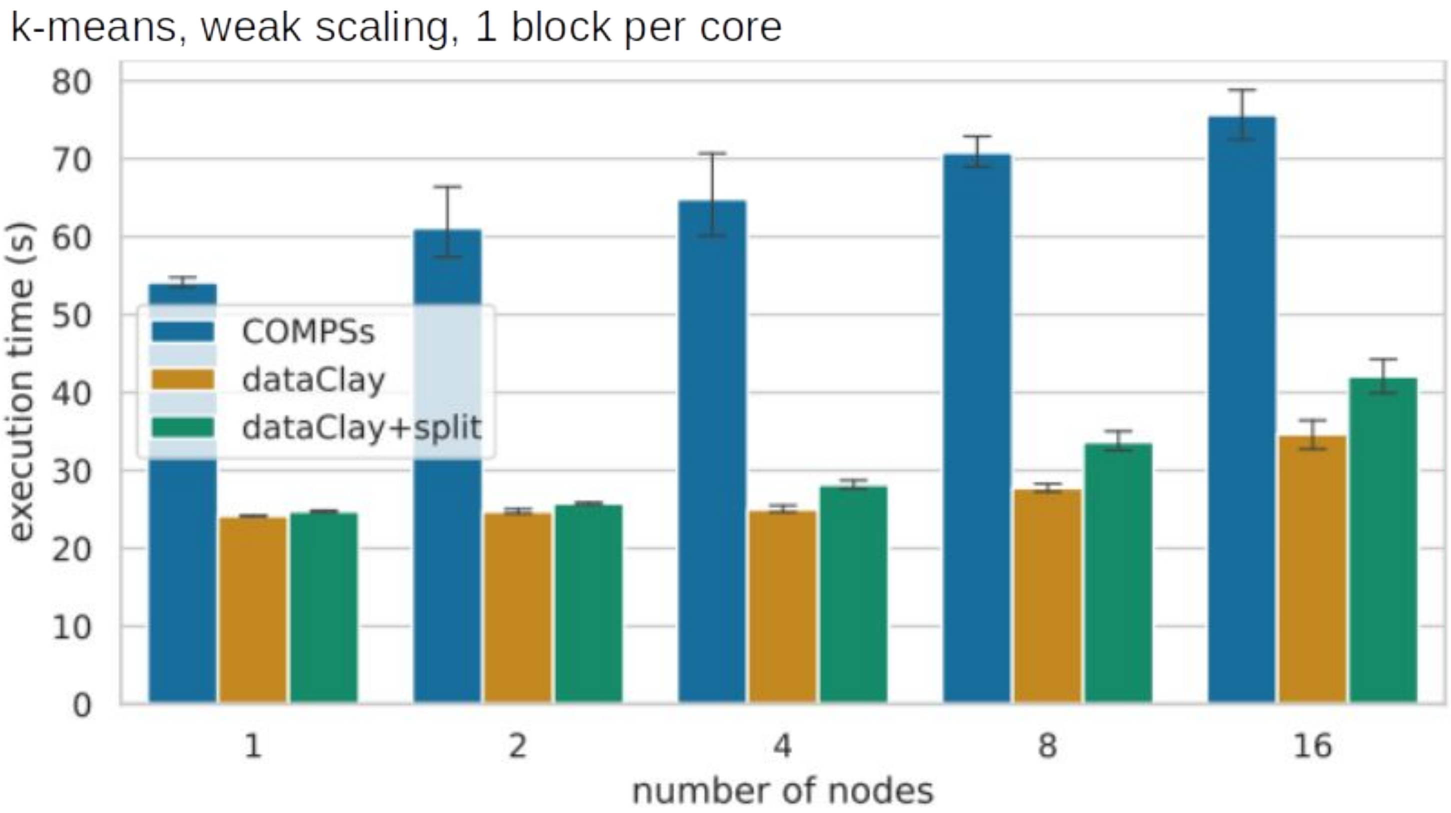
- Reduces the number of tasks
  - Less work for the scheduler
  - Less runtime overheads for task invocation
- Groups blocks in the same node
  - Preserves locality between tasks and blocks
  - Data locality also for returns / reduction



Results (I)


k-means, weak scaling, 48 blocks per core
COMPSs
dataClay
dataClay+split
4200s >>
execution time (s)
1000
800
600
400
200
0
1
2
4
8
16
number of nodes

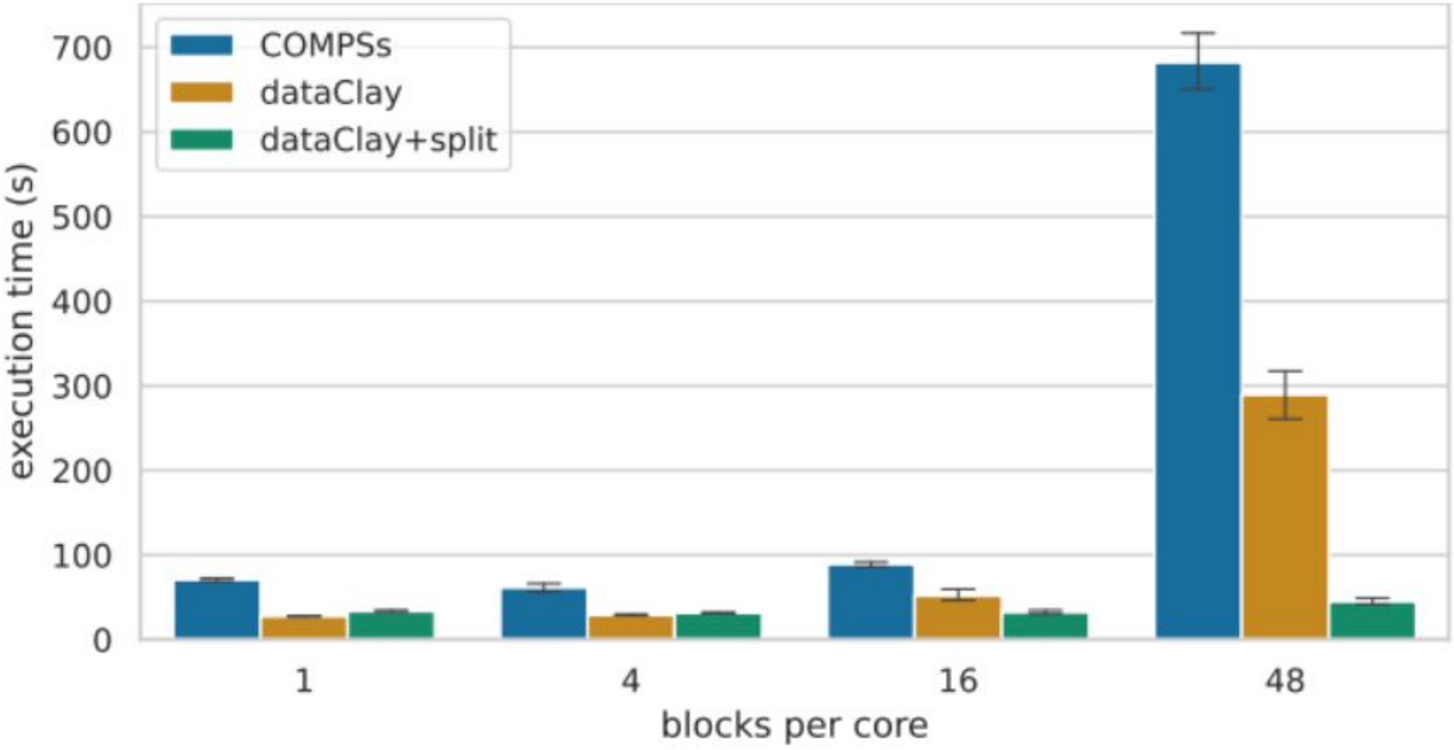
Results (II)



k-means, weak scaling, 1 block per core
80
70
60
50
40
30
20
10
0
execution time (s)
COMPSs
dataClay
dataClay+split
1
2
4
8
16
number of nodes

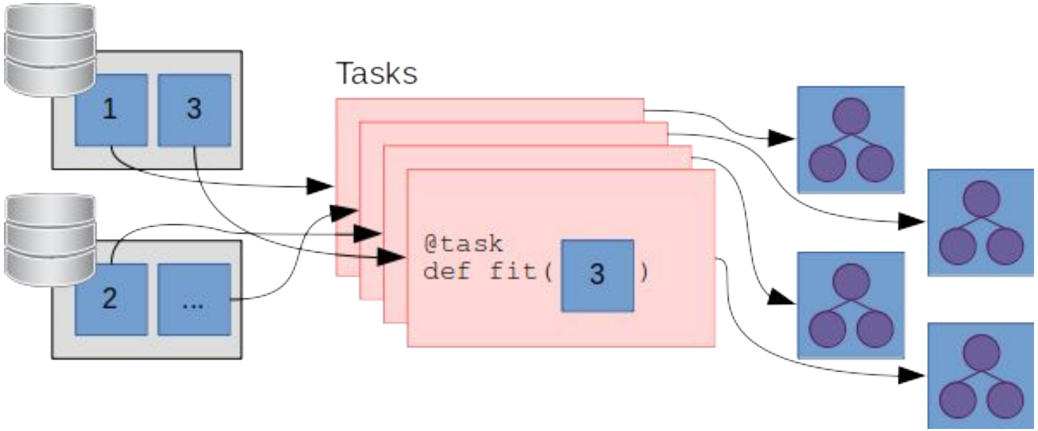
Results (III) - Sensitivity to fragmentation

kmeans execution on 8 nodes, fixed dataset size. Block size changes



Algorithmic improvements: kNN Use Case (I)

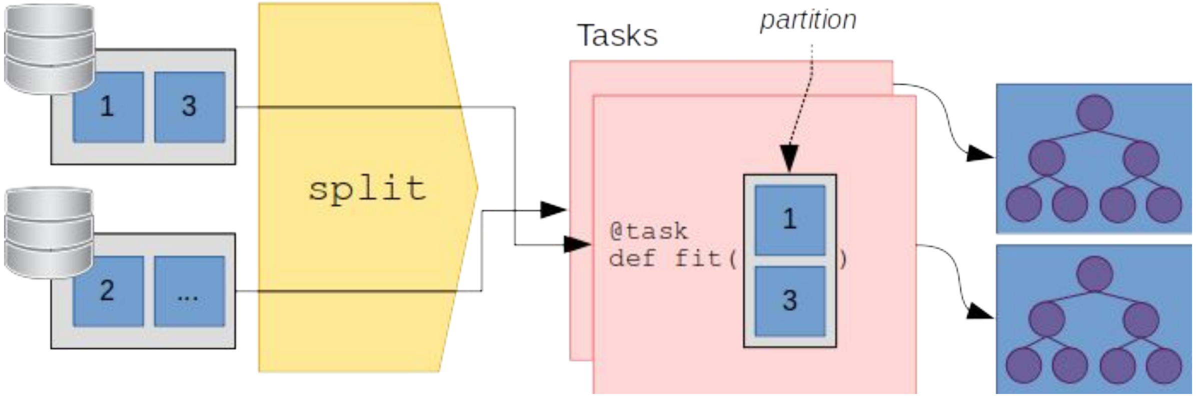
***k*-Nearest Neighbors algorithm starts with a fit procedure that builds tree data structures:**



The size of the trees depends on the size of the block (granularity)

Algorithmic improvements: kNN Use Case (II)

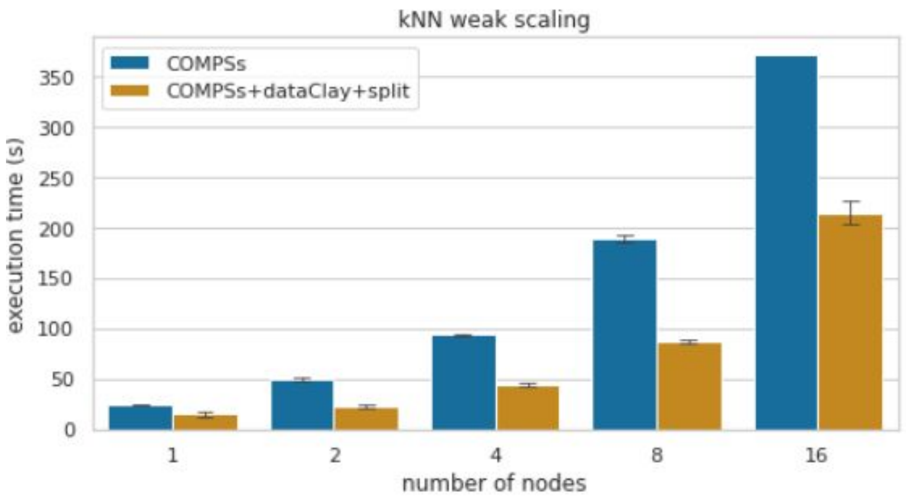
By using the split, the tree data structures can be built bigger:



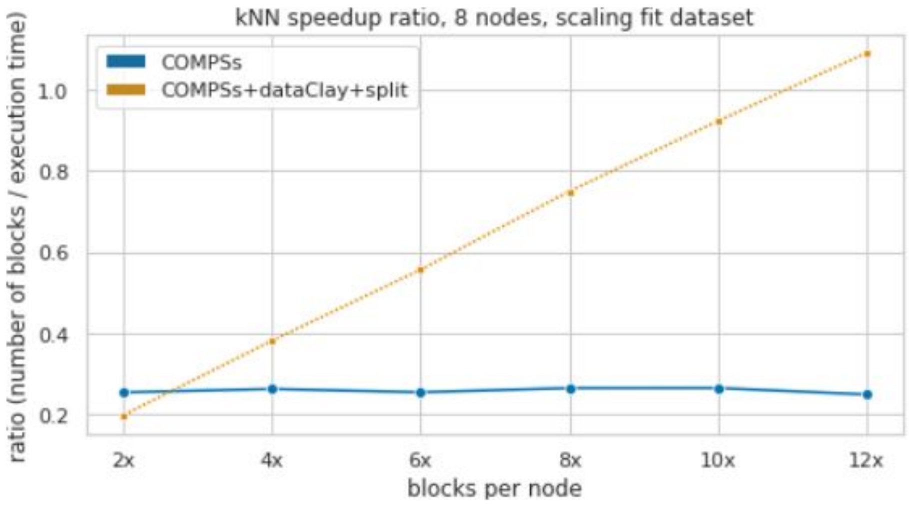
The split preserves locality, each tree is built from blocks within the same node.

Having bigger trees results in algorithmic improvements.

kNN Results



kNN weak scaling
COMPSs
COMPSs+dataClay+split
execution time (s)
350
300
250
200
150
100
50
0
1
2
4
8
16
number of nodes



kNN speedup ratio, 8 nodes, scaling fit dataset
COMPSs
COMPSs+dataClay+split
ratio (number of blocks / execution time)
1.0
0.8
0.6
0.4
0.2
2x
4x
6x
8x
10x
12x
blocks per node

Conclusions

- **When in doubt, use split**
  - If your dataset is fragmented, you will get benefits
    - Even for compute bound applications!
- **If size of blocks is optimal, you pay overhead with no benefits**
  - If you KNOW your optimal block size, set it and avoid the split
    However, that may prove difficult or unfeasible
- **Your intermediate data structures may benefit from the split**
  - Algorithmic knowledge is required
  - Benefits can be substantial, O(log(n)) vs O(n)

Thank you

eFlows4HPC

Enabling dynamic and Intelligent workflows in the future EuroHPC ecosystem

www.eFlows4HPC.eu

@eFlows4HPC

eFlows4HPC Project

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Germany, France, Italy, Poland, Switzerland, Norway.