Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Edge-to-end data workflows

Rosa M. Badia, Javier Conejero, Daniele Lezzi, Raül Sirvent

23/05/2023

Solid Earth and Geohazards in the Exascale Era

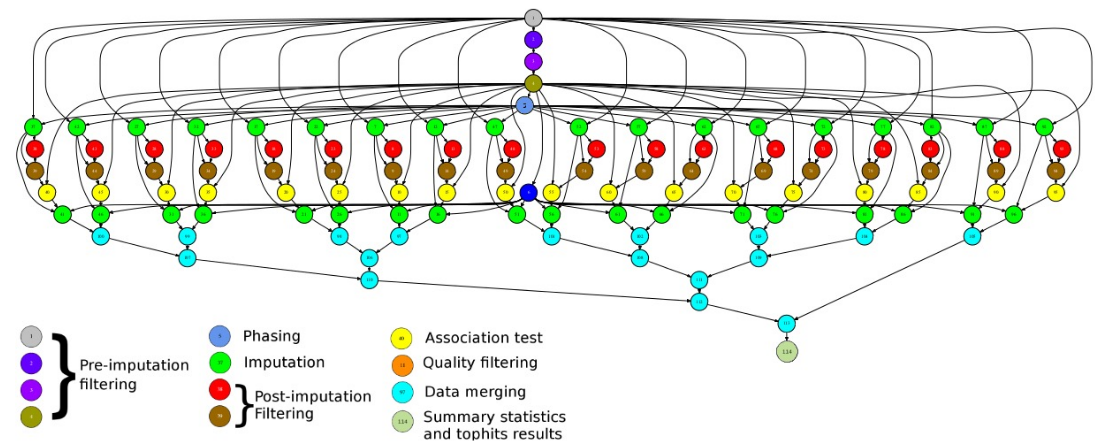
Agenda

9:00 – 11:00	Introduction by Masters Presentation by students (30 min each in total, including discussion and interaction)
11:00 - 11:30	Break
11:30 - 13:30	Presentation by students (30 min each in total, including discussion and interaction)
13:30 - 14:30	Lunch break
14:30 - 16:30	Master pitch presentation Master Class Plenary Session (more collective discussion) Lessons learnt
16:30 – 17:00	Break
17:00 - 19:00	Hands-on

Main element: Workflows in PyCOMPSs

- Sequential programming, parallel execution
- General purpose programming language + annotations/hints
  - To identify tasks and directionality of data
- Builds a task graph at runtime that express potential concurrency
- Tasks can be sequential and parallel (threaded or MPI)
- Offers to applications the illusion of a shared memory in a distributed system
  - The application can address larger data than storage space: support for Big Data apps
  - Support for persistent storage
- Agnostic of computing platform
  - Enabled by the runtime for clusters, clouds and container managed clusters

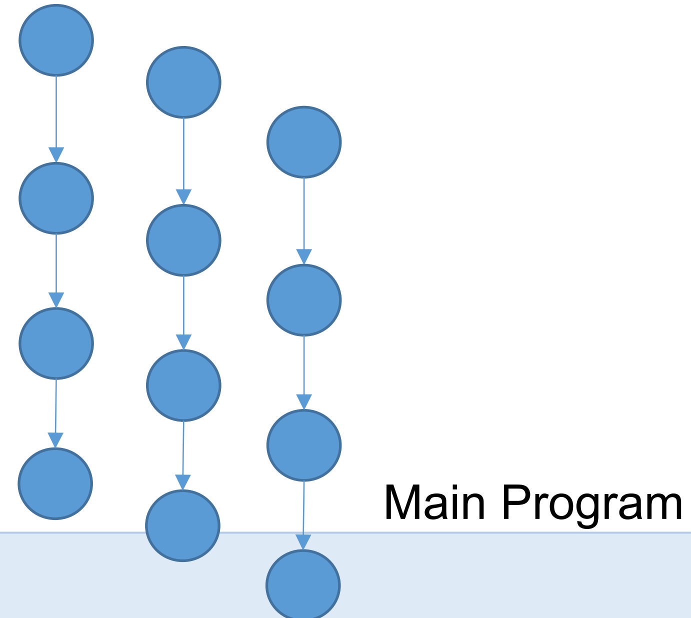
```
@task(c=INOUT)
def multiply(a, b, c):
      c += a*b
```



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

PyCOMPSs syntax

- Use of **decorators** to annotate tasks and to indicate arguments directionality
- Small API for data synchronization



Tasks definition

```
@task(c=INOUT)
def multiply(a, b, c):
    c += a*b
```

Main Program

```
initialize_variables()
startMulTime = time.time()
for i in range(MSIZE):
   for j in range(MSIZE):
      for k in range(MSIZE):
         multiply (A[i][k], B[k][j], C[i][j])
compss_barrier()
mulTime = time.time() - startMulTime
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Synchronization

- Main program and tasks do not share the same memory spaces
- The synchronization compss_wait_on waits for tasks generating the parameter to be finished and moves the data from the remote node to the node where the main program is executed:

```
a = compute (b)
#compute is a task, here we can not check the value of a
...
a = compss_wait_on (a)
#here we can check the value of a
if a:
    ...
```

- Tasks can be also synchronized with a barrier

```
startMulTime = time.time()
for i in range(SIZE):
    compute (A[i], B[i])
compss_barrier()
mulTime = time.time() - startMulTime
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Other decorators: Tasks' constraints

- Constraints enable to define HW or SW features required to execute a task
  - Runtime performs the match-making between the task and the computing nodes
  - Support for multi-core tasks and for tasks with memory constraints
  - **Support for heterogeneity on the devices in the platform**

```
@constraint (MemorySize=6.0, ProcessorPerformance="5000", ComputingUnits="8")
@task (c=INOUT)
def myfunc(a, b, c):
    ...
```

```
@constraint (MemorySize=1.0, ProcessorType ="ARM", )
@task (c=INOUT)
def myfunc_other(a, b, c):
    ...
```

Failure Management

- Interface that enables the programmer to give hints about failure management

```
@task(file_path=FILE_INOUT, on_failure='CANCEL_SUCCESSORS',
time_out='$task_timeout')
def task(file_path):
    ...
    if cond :
        raise Exception()
```

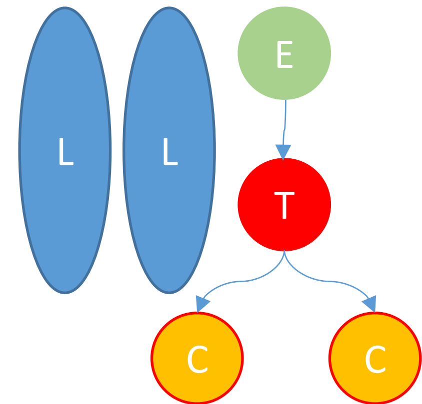
- Options: RETRY, CANCEL_SUCCESSORS, FAIL, IGNORE
- Implications on file management:
    - I.e., on IGNORE, output files: are generated empty
- **Possibility of ignoring part of the execution of the workflow, for example if a task fails in an unstable device**
- **Opens the possibility of dynamic workflow behaviour depending on the actual outcome of the tasks**

- Tasks can raise exceptions

```
@task(file_path=FILE_INOUT)
def comp_task(file_path):
    ...
    raise COMPSsException("Exception raised")
```

- Combined with groups of tasks enables to cancel the group of tasks on the occurrence of an exception

```
def test_cancellation(file_name):
    try:
        with TaskGroup('failedGroup'):
            long_task(file_name)
            long_task(file_name)
            executed_task(file_name)
            comp_task(file_name)
            cancelledTask(FILE_NAME);
            cancelledTask(FILE_NAME)

except COMPSsException:
        print("COMPSsException caught")
        write_two(file_name)
```



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Other decorators: linking with other programming models

- A task can be more than a sequential function
  - A task in PyCOMPSs can be sequential, multicore or multi-node
  - External binary invocation: wrapper function generated automatically
  - Supports for alternative programming models: MPI and OmpSs
- Additional decorators:
  - @binary(binary="app.bin")
  - @mpi(binary="mpiApp.bin", runner="mpirun", processes=8)
  - @ompss(binary="ompssApp.bin")
- Can be combined with the @constraint and @implement decorators


```
@binary(binary="app.bin", workingDir="/myApp")
@task()
def func(l):
    pass
```

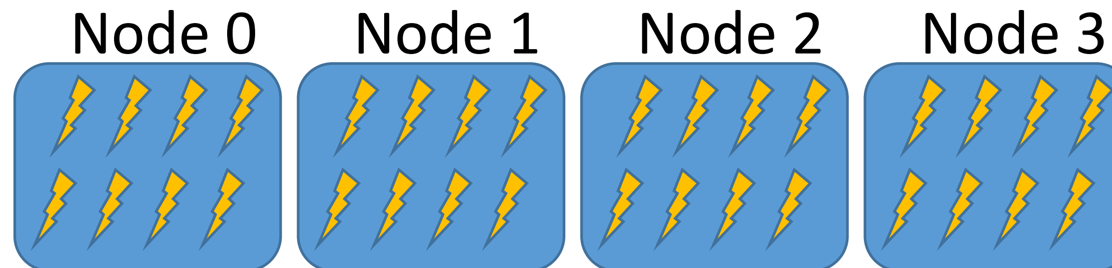
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Support for MPI tasks

- Resource manager aware of multi-node tasks

```
@mpi (runner="mpirun", processes= "32", processes_per_node=8)
@task (returns=int, stdOutFile=FILE_OUT_STDOUT, stdErrFile=FILE_OUT_STDERR)
def nems(stdOutFile, stdErrFile):
    pass
```

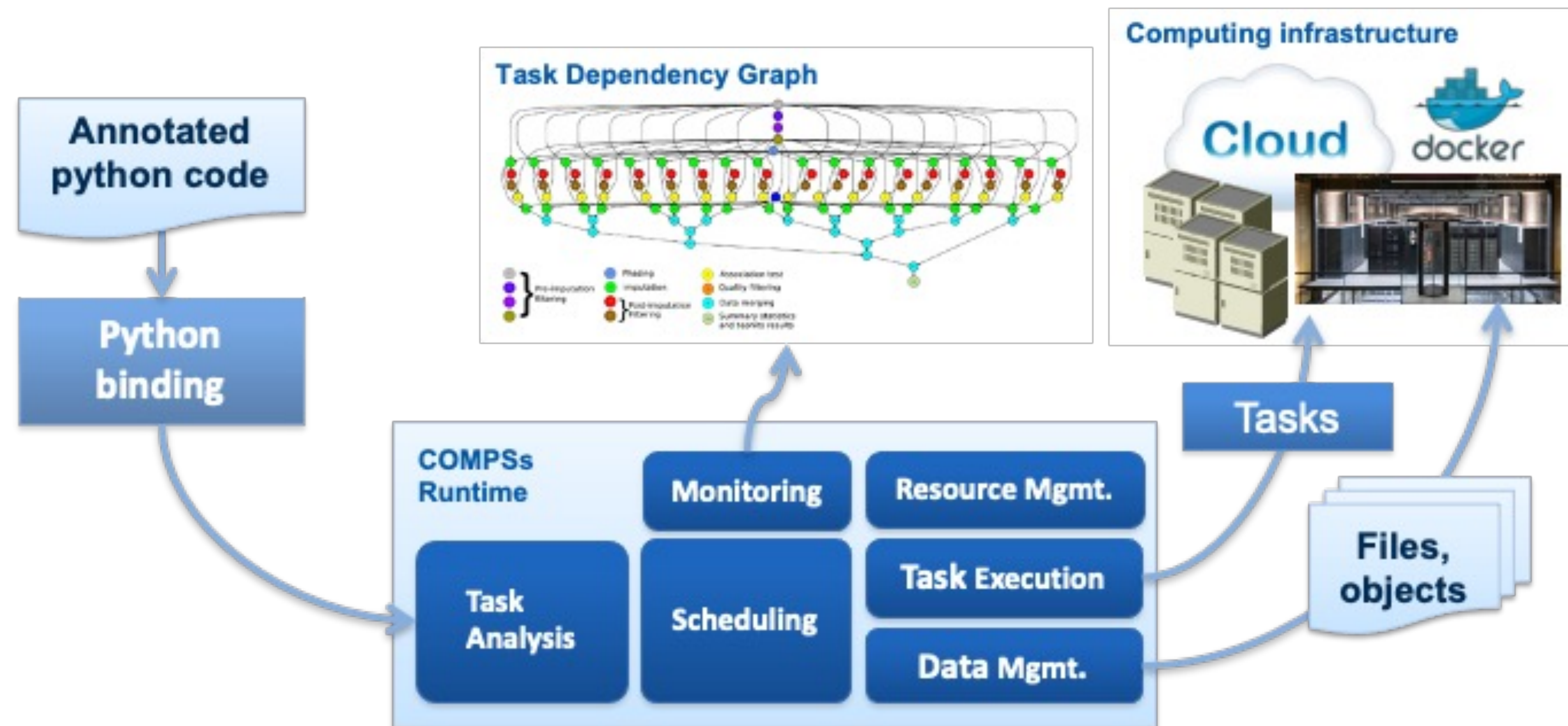
Launches MPI execution with
32 processes
8 processes per node



BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

PyCOMPSs runtime

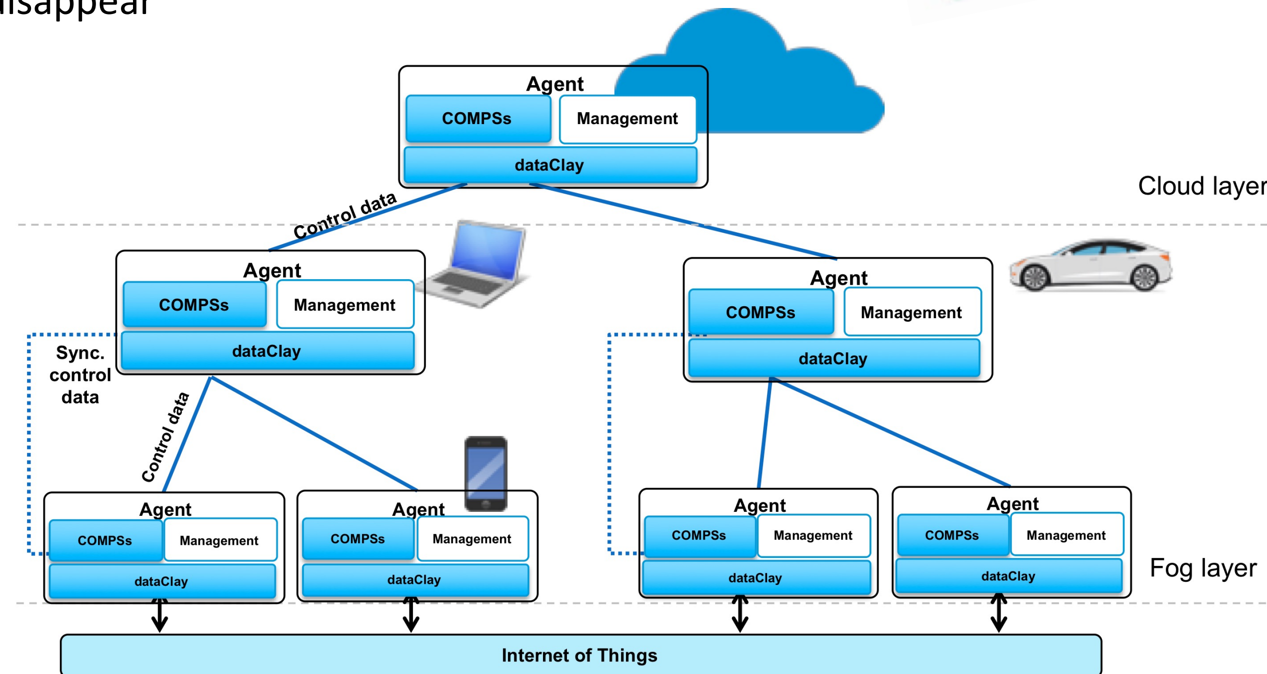
- Runtime deployed as a distributed master-worker
- All data scheduling decisions and data transfers are performed by the runtime
- Support for elasticity



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

COMPSs in a fog-to-cloud architecture

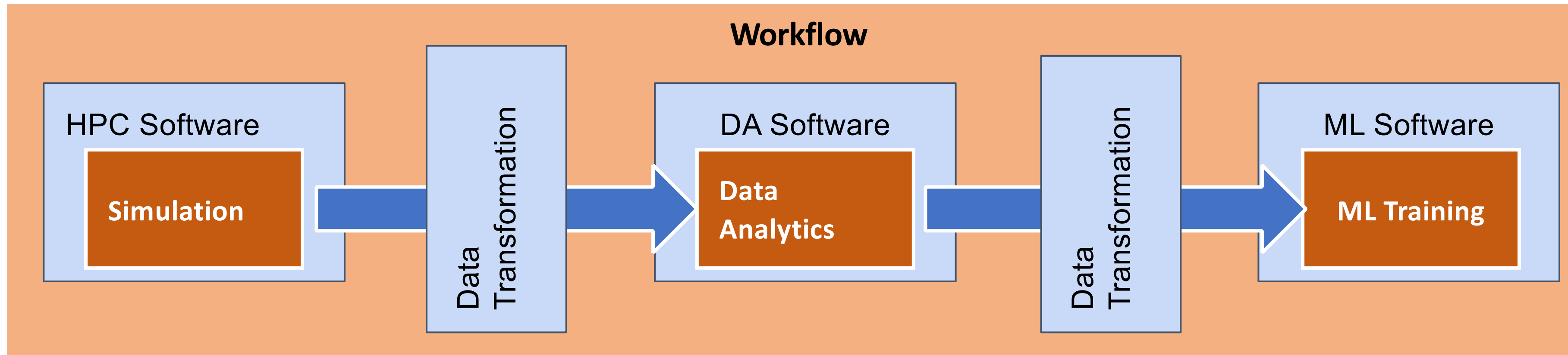
- **Decentralized** approach to deal with large amounts of data
- New COMPSs runtime handles distribution, parallelism and heterogeneity
- Runtime deployed as a microservice in an agent:
    - Agents are independent, can act as master or worker in an application execution, agents interact between them
    - Hierarchical structure
- Data managed by dataClay, in a federated mode
    - Support for data recovery when fog nodes disappear
- Fog-to-fog and Fog-to-cloud



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Interfaces to integrate HPC/DA/ML

eFlows4HPC
Enabling dynamic and Intelligent workflows in the future EuroHPC ecosystem

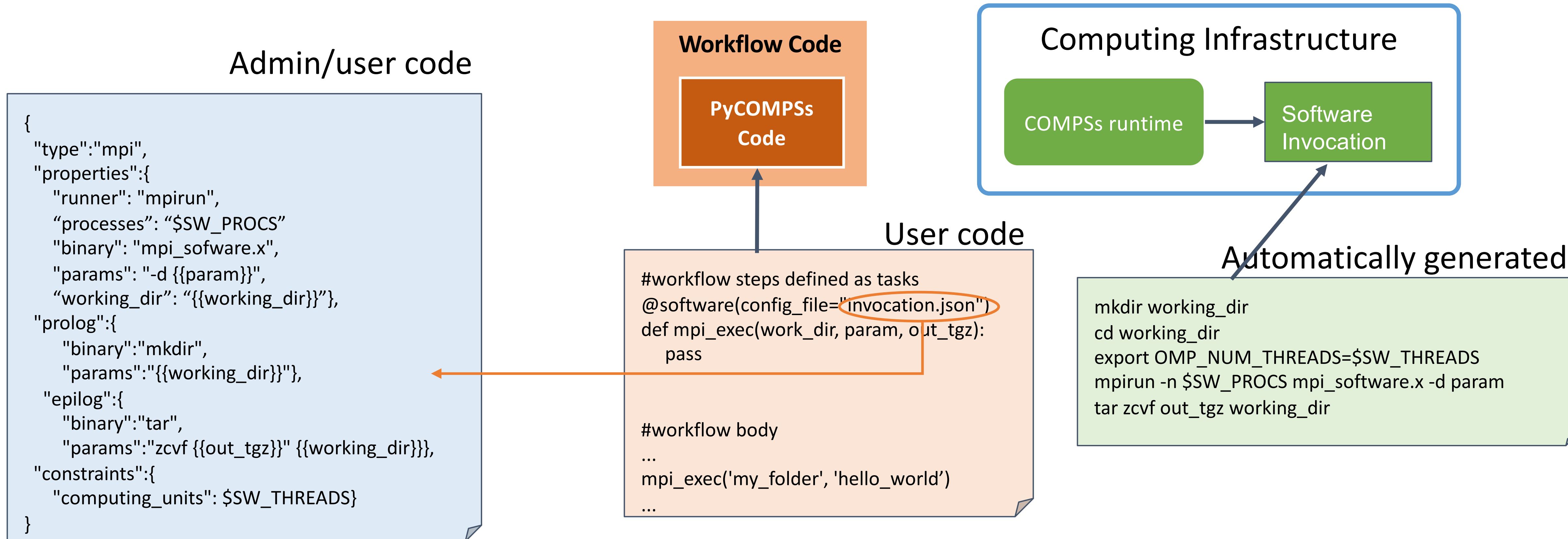


- Goal:
  - Reduce the required glue code to invoke multiple complex software steps
  - Developer can focus in the functionality, not in the integration
  - Enables reusability
- Two paradigms:
  - Software invocation
  - Data transformations

```
#workflow steps defined as tasks
@data_transformation (input_data, transformation description)
@software (invocation description)
def data_analytics (input_data, result):
    pass

#workflow body
simulation (input_cfg, sim_out)
data_analytics (sim_out, analysis_result)
ml_training (analysis_result, ml_model)
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Software Invocation description



Software invocation description
Stored in software catalog

- Converts a Python function of a software invocation to a PyCOMPSs task
- Takes information from the description in json
- Enables reuse in multiple workflows

eFlows4HPC
Enabling dynamic and Intelligent workflows in the future EuroHPC ecosystem

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Data transformations

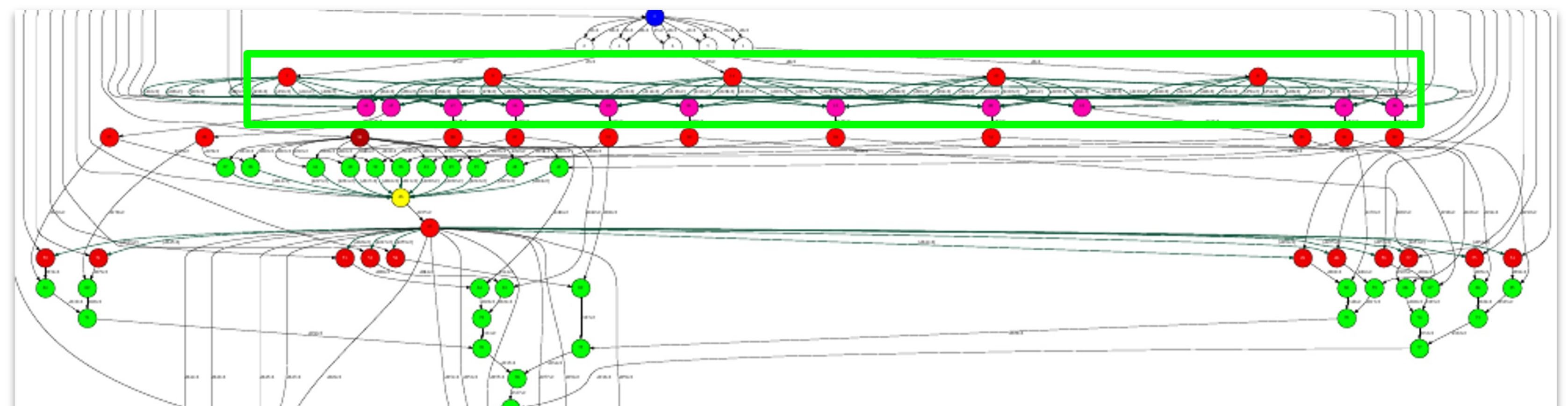
- A data transformation changes the data without requiring extra programming from the developer

Admin/user code

```
def load_blocks_rechunk(blocks, shape, block_size, new_block_size):
    ...
    SnapshotMatrix = load_blocks_array (final_blocks, shape, block_size);
    return SnapshotMatrix
```

User code

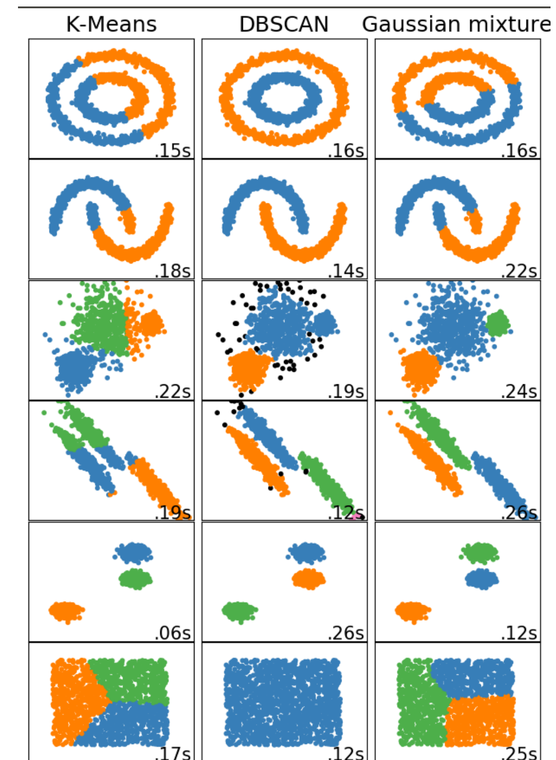
```
@dt("blocks", load_blocks_rechunk, shape=expected_shape, block_size=simulation_block_size,
 new_block_size=desired_block_size, is_workflow=True)
@software(config_file = SW_CATALOG + "/dislib/dislib.json")
def rSVD(blocks, desired_rank=30):
    u,s = rsvd(blocks, desired_rank, A_row_chunk_size, A_column_chunk_size)
    return u
```



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Dislib: parallel machine learning

- dislib: Collection of machine learning algorithms developed on top of PyCOMPSs
  - Unified interface, inspired in scikit-learn (fit-predict)
  - Based on a distributed data structure (ds-array)
  - Unified data acquisition methods
  - Parallelism transparent to the user – PyCOMPSs parallelism hidden
  - Open source, available to the community
- Provides multiple methods:
  - data initialization
  - Clustering
  - Classification
  - Model selection, ...



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

dislib.bsc.es

DISLIB | Distributed Computing Library

Agenda for presentations

- Linus Walter
- Marisol Monterrubio
- Rut Blanco
- Claudia Abril
- Natalia Zamora
- Alejandra Guerrero
- Marc Martínez Sepúlveda

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Thank you

eFlows4HPC
Enabling dynamic and Intelligent workflows
in the future EuroHPC ecosystem

Tasks in container images

- Goal: enable tasks embedded in container images
- New @container decorator to be used together with the task annotation
- Also supports user-defined tasks

```
@container(engine="DOCKER", image="ubuntu")
@binary(binary="ls")
@task()
def task_binary_empty():
    pass
```

```
@container(engine="DOCKER", image="compss/compss")
@task(returns=1, num=IN, in_str=IN, fin=FILE_IN)
def task_python_return_str(num, in_str, fin):
    print("Hello from Task Python RETURN")
    print("- Arg 1: num -- " + str(num))
    print("- Arg 1: str -- " + str(in_str))
    print("- Arg 1: fin -- " + str(fin))
    return "Hello"
```

MPMD applications

- The @mpmd_mpi decorator can be used to define Multiple Program Multiple Data (MPMD) MPI tasks

```
@mpmd_mpi(runner="mpirun", working_dir = {{working_dir_exe}},
          programs=[{binary="fesom.x", processes = "$FESOM_PROCS" },
                   {binary="oifs", args="-v ecmwf -e awi3", processes = "$OIFS_PROCS" },
                   {binary="rnfma", processes = "$RNFMA_PROCS"}])
@task(log_file={Type:FILE_OUT, StdIOStream:STDOUT}, working_dir_exe=DIRECTORY_INOUT)
def esm_simulation(log_file, working_dir_exe):
   pass
```

- As a result of the @mpmd_mpi annotation, the following commands will be generated:

```
> cd working_dir_exe; mpirun -n $FESOM_PROCS fesom.x : \
     -n $OIFS_PROCS oifs -v ecmwf -e awi3 : -n $RNFMA_PROCS rnfma
```

Afternoon discussion

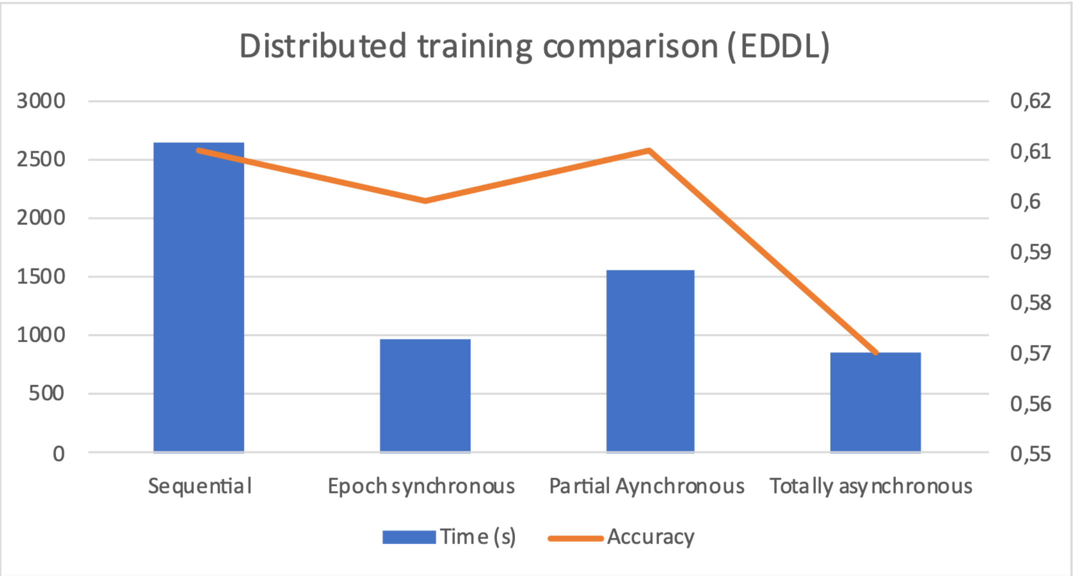
- Topics
  - Distributed training with dislib
  - Parallel execution with tensorflow
  - Cybershake implementation
  - Usage of RF dislib model from scikitlearn
  - Block size
  - Time series with scikit-learn
  - PyCOMPSs with containers
  - CIC service and preliminary results
  - PyCOMPSs with microservices
  - Version control
  - Sample workflow in PyCOMPSs

Distributed training with dislib

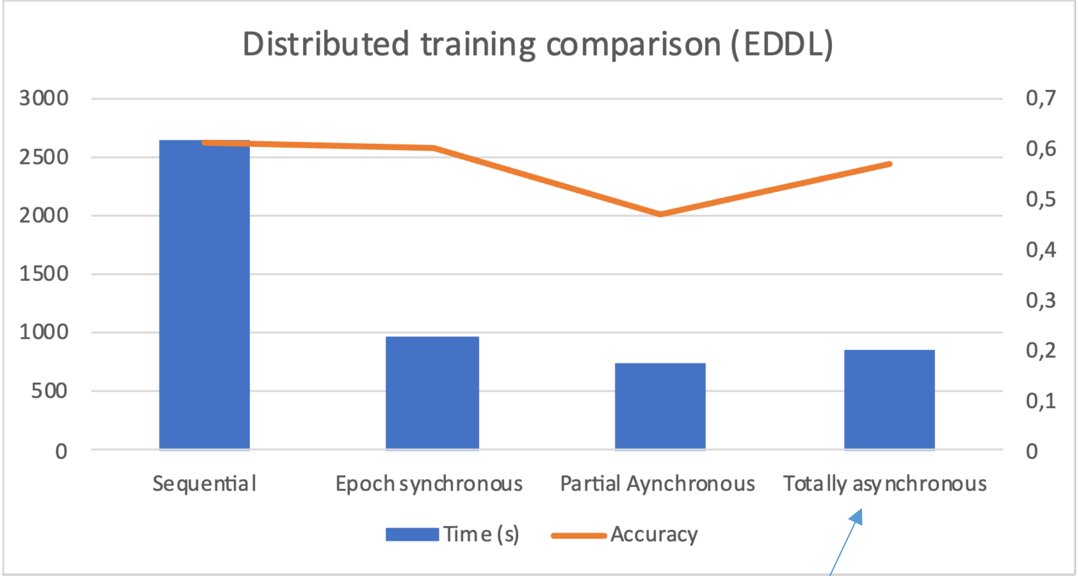
- Three different strategies for training:
  - **Epoch-synchronous training:** each worker performs the training in parallel, and after each epoch, a synchronisation point occurs. After this, new weights are computed and assigned to the neural network.
    - Data shuffling, two options:
      - random block exchange between workers combined with a local shuffle inside the worker
      - total data shuffling (all data is exchanged between all blocks in all workers).
        - This strategy should be equivalent to a sequential training if total shuffling is used.
    - The drawback of this method is that performance can be reduced due to the global synchronisation point appearing after each epoch. An alternative version has also been implemented, where the global weights are only updated after a group of epochs (instead of after each epoch).
  - **Total asynchronous training:** global weights are only updated after all epochs have been completed.
    - Once a worker completes one epoch, it shuffles the data and proceeds with a new one.
    - For the data shuffle the same two options described in the epoch-synchronous case are available.
  - **Partial asynchronous training:** when a worker completes an epoch, partially updates the global weights with the local results.
    - Implemented by means of the “commutative” clause of PyCOMPSs which enables only one task at a time to perform this update.
    - This strategy performs an incremental update of the global weights that are available for the next iteration of those workers that start a new training. This avoids per-epoch synchronisations (case 1) with the goal of reducing the total

Results with EDDL

- Data set
  - Cifar-10
  - 160-170 MB
  - Split into 4 fragments



Partial asynchronous with 50 Epochs, others with 30 Epochs



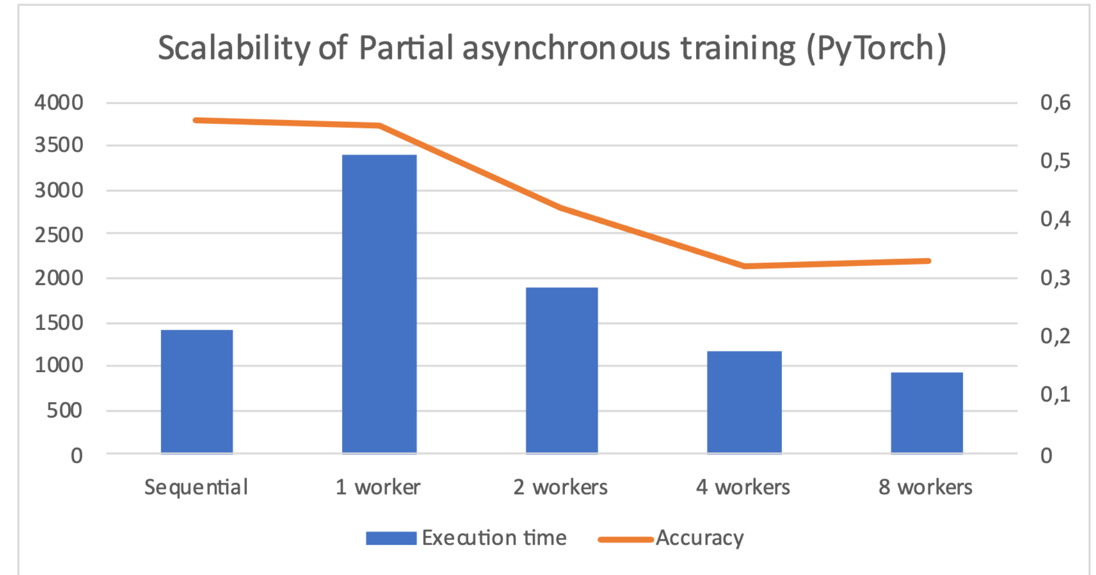
All trainings performed with 30 Epochs

Synchronizing every 4 epochs

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Results with PyTorch

- Data set
  - Imagenette Dataset
  - 1.5 GB
  - Dataset split in 24 tensors



Executed with 1 GPU per worker and node

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Tensorflow

- https://www.tensorflow.org/guide/distributed_training

Dislib RF model -> scikit learn

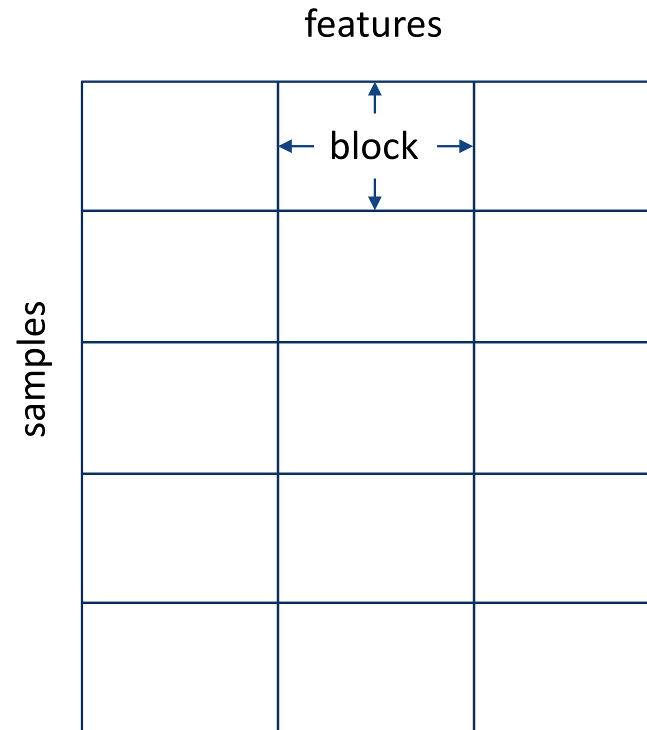
- Apparently not compatible

Dislib block size

- https://www.overleaf.com/project/63be8f7eb69dd8b7684e3c6e

Distributed array (ds-array)

- 2-dimensional structure (i.e., matrix)
  - Divided in blocks (NumPy arrays)
- Works as a regular Python object
  - But not stored in local memory!
- Methods for instantiation and slicing with the same syntax of numpy arrays:
  - Internally parallelized with PyCOMPSs:
  - Loading data (e.g. from a text file)
  - Indexing (e.g., x[3], x[5:10]
  - Operators (e.g., x.min(), x.transpose())
- ds-arrays can be iterated efficiently along both axes
- Samples and labels can be represented by independent distributed arrays



Times series with dislib

- Time series with dislib?
- Call scikit learn with PyCOMPSs per each station?

Tasks in container images

- Goal: enable tasks embedded in container images
- New @container decorator to be used together with the task annotation
- Also supports user-defined tasks

```
@container(engine="DOCKER", image="ubuntu")
@binary(binary="ls")
@task()
def task_binary_empty():
    pass
```

```
For i in range (0,3):
    task_binary_empty():
```

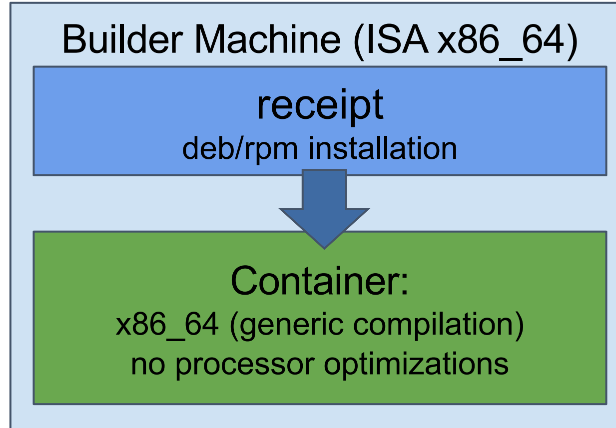
```
@container(engine="DOCKER", image="compss/compss")
@task(returns=1, num=IN, in_str=IN, fin=FILE_IN)
def task_python_return_str(num, in_str, fin):
    print("Hello from Task Python RETURN")
    print("- Arg 1: num -- " + str(num))
    print("- Arg 1: str -- " + str(in_str))
    print("- Arg 1: fin -- " + str(fin))
    return "Hello"
```


BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

```python
# ## TASK SELECTION ## #
@container(engine="UDOCKER",
           image="compss/compss")
@task(c=INOUT)
def multiply(a, b, c):
    import numpy as np
    c += a*b


# ## MAIN PROGRAM ## #

if __name__ == "__main__":

    args = sys.argv[1:]

    MSIZE = int(args[0])
    BSIZE = int(args[1])

    A = []
    B = []
    C = []

    initialize_variables()

    for i in range(MSIZE):
        for j in range(MSIZE):
            for k in range(MSIZE):
                multiply(A[i][k], B[k][j], C[i][j])
```

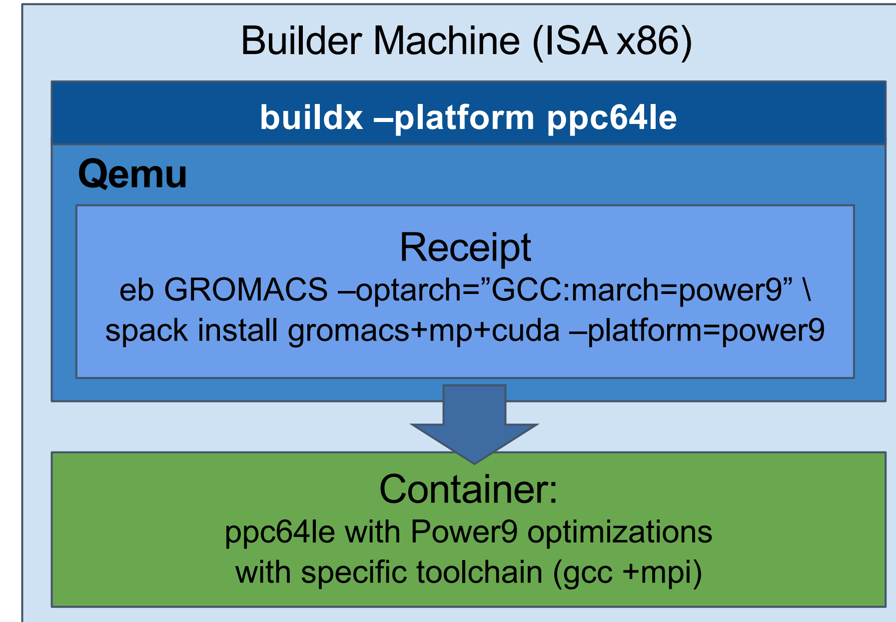
Containers and HPC

Standard container image creation



- **Simplicity for deployment**
  - Just pull or download the image
- **Trade-Off performance/portability**
  - Architecture Optimizations
- **Accessing Hardware from Containers**
  - MPI Fabric /GPUs
  - Host-Container Version Compatibility needed

eFlows4HPC approach

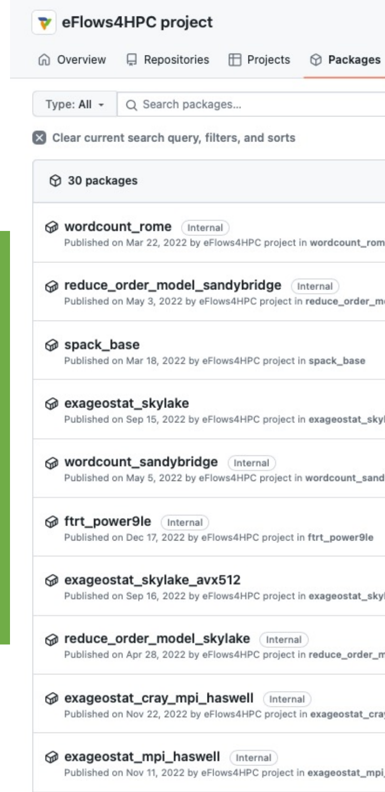


- **Methodology to allow the creation containers for specific HPC system**
  - Leverage HPC and Multi-platform container builders
- **It is hard to do by hand but let's automate!**

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

HPC Ready Containers

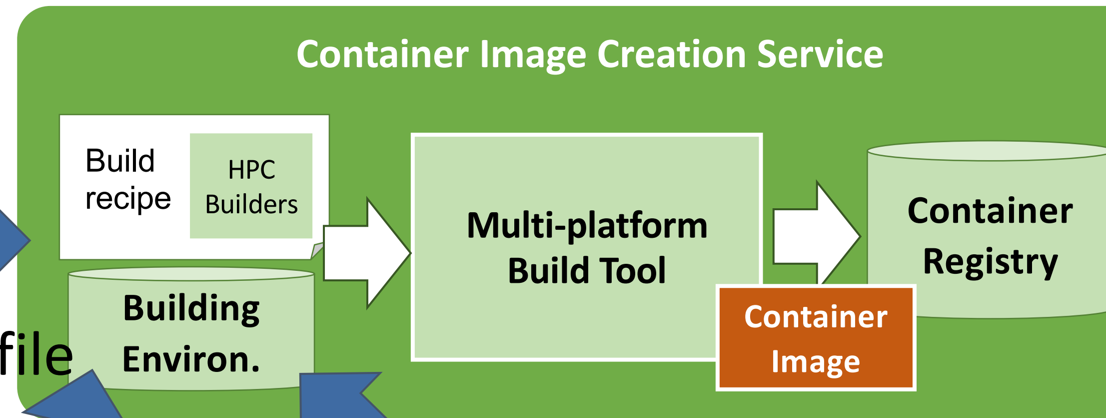
eFlows4HPC



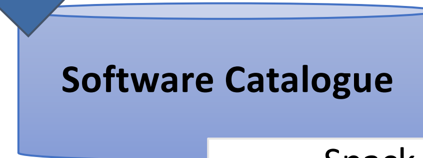
Workflow step + target system

```
1  {
2      "machine": {
3          "platform": "linux/amd64",
4          "architecture": "sandybridge",
5          "container_engine": "singularity"
   ↪
6          },
7      "workflow":"rom_pillar_I",
8      "step_id" :"reduce_order_model"
9  }
```

json request file



Container components (spack Environment)




```
1  spack:
2          specs:
3                  - compss
4                  - py-dislib
5                  - kratos apps=LinearSolversApplication, RomApplication
```

spack.yml

Spack specs
Installation Description

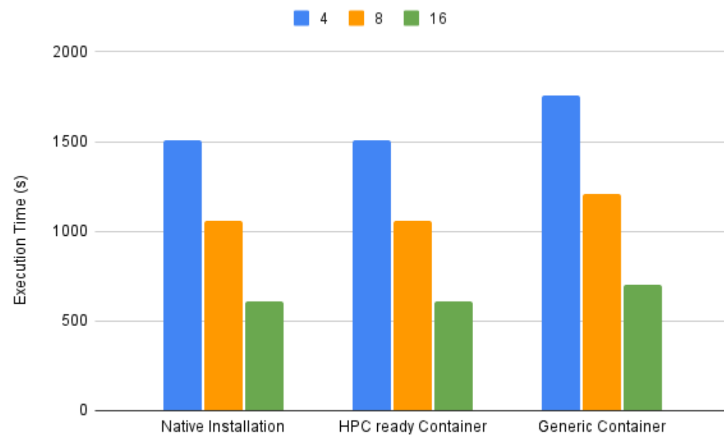
package.py

*Ejarque, J and Badia, R. "Automatizing the creation of specialized high-performance computing containers." IJHPCA (2023), doi.org/10.1177/10943420231165729.

HPC-Ready Containers

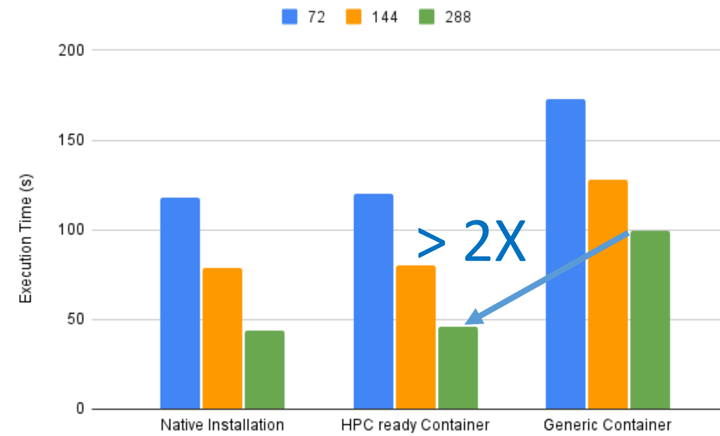
Kratos Multiphysics (shared memory)

Nord3



core counts / task

FESOM2 (MPI)



MPI processes

MN4

6 12 24 48

1500
1000
500
0
Execution Time (s)
~ 1.4X
Native Installation
HPC ready Container
Generic Container

core counts / task

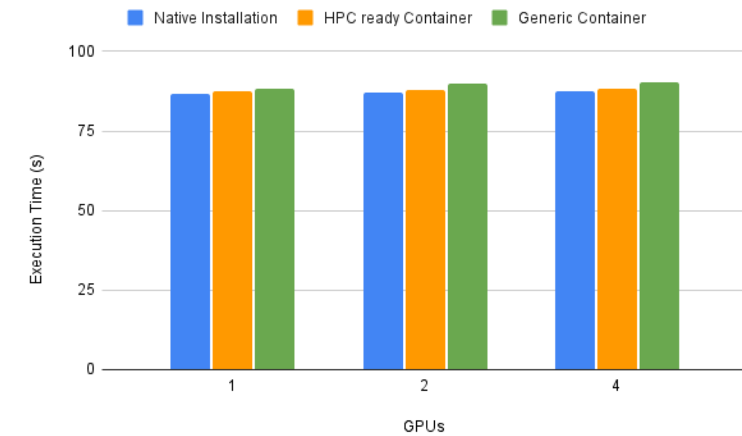
72 144 288

150
100
50
0
Execution Time (s)
Native Installation
HPC ready Container
Generic Container

MPI processes

CTE-Power 9

Tsunami-HySEA



#GPUs

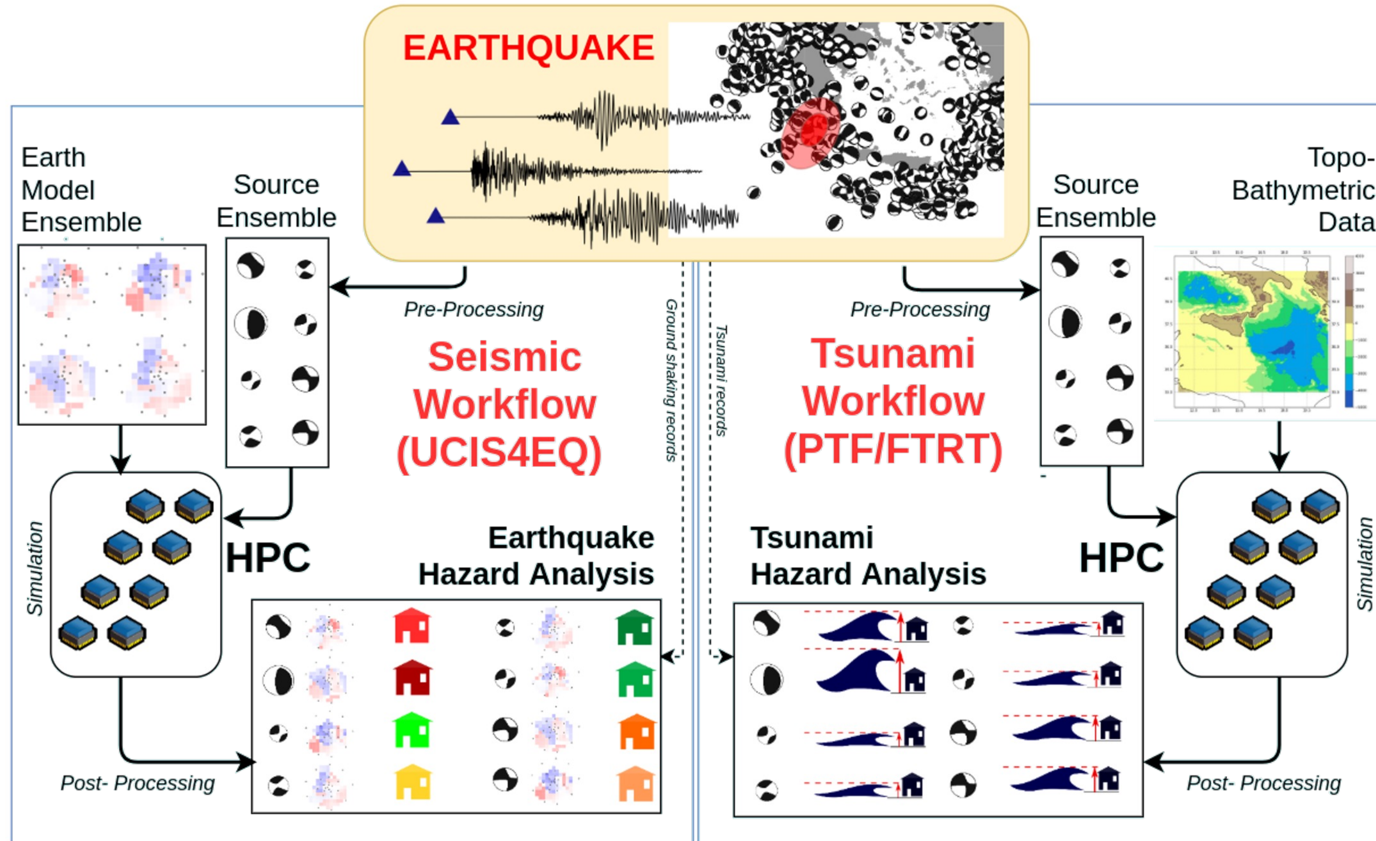
Most of the execution performed in the GPU

*Ejarque, J and Badia, R. "Automatizing the creation of specialized high-performance computing containers." IJHPCA (2023), doi.org/10.1177/10943420231165729

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Pillar III: Urgent computing for natural hazards

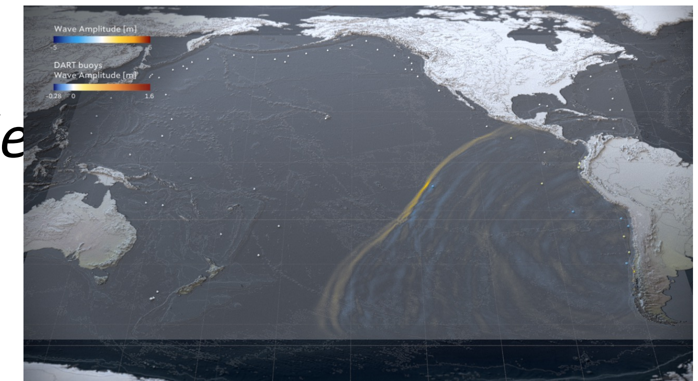
eFlows4HPC



Pillar III explores the modelling of natural catastrophes:

- Earthquakes and their associated tsunamis shortly after such an event is recorded
- Use of AI to estimate intensity maps
- Use of DA and AI tools to enhance event diagnostics
- Areas: Mediterranean basin, Mexico, Iceland and Chile

Tsunami-HySEA GPU-based code



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

ETH zürich

ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA

PSNC

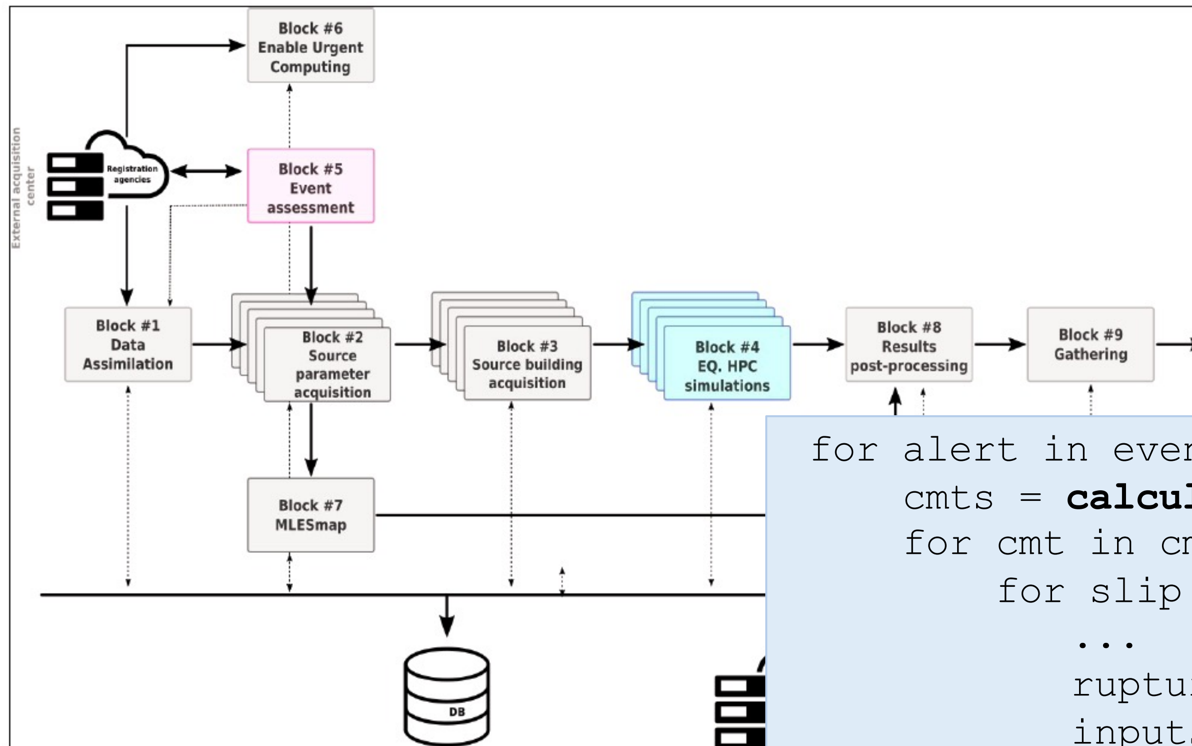
UNIVERSIDAD DE MÁLAGA

NGI

Pillar III: UCIS4EQ workflow: http services as tasks

eFlows4HPC

Urgent Computing Integrated Services for Earthquakes: UCIS4EQ




```
@http(request="POST", resource="SalvusRun", ...')
@task(returns=1)
def run_salvus(event_id, trial, input, resources):
    """
    """
    pass
@http(request="POST", resource="cmt",...)
@task(returns=1)
def calculate_cmt(alert, event_id, domain, precmt):
    """
    """
    pass
...
```

```
for alert in event['alerts']:
    cmts = calculate_cmt(alert, eid, domain, precmt)
    for cmt in cmts.keys():
        for slip in range(1, region['GPSetup']['trials']+1)
            ...
            rupture = compute_graves_pitarka(eid, alert, ...)
            inputs = build_input_parameters( eid, alert, ...)
            salvus_inputs = build_salvus_parameters( eid, path, ...)
            result = run_salvus( eid, path, ...)
            all_results.append(run_salvus_post(eid, result, ...))
result = run_salvus_plots(eid, basename, domain, resources)
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

- Sample workflow
  - https://docs.google.com/document/d/1QKolZoUi3OwkWppvK-wCxfwOZIjCxT1-/edit

BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación