PyCOMPSs v3.4 – Cheat Sheet Advanced Features

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Barcelona Supercomputing Center, Workflows and Distributed Computing Group

Constraints

Find the proper resource for a task

```
from pycompss.api.constraint import constraint

@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):
    ...
```

Parallel Tasks

Include multi-core or multi-node tasks. Decorators @binary, @ompss, @mpi, @mpmd_mpi, @multinode, @Julia and

```
from pycompss.api.mpi import mpi

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

```
from pycompss.api.binary import binary

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

Reduction

Merge multiple tasks' results

```
from pycompss.api.reduction import reduction

@reduction(chunk_size="2")
@task(returns=1, col=COLLECTION_IN)
def myreduction_1(col):
    r = 0
    for i in col:
        += i
    return r
```

HTTP tasks

Define a task as an invocation to an HTTP service

```
from pycompss.api.http import http

@http(service_name="service_1", request="POST", resource="post_json/",
      payload="{{payload}}", payload_type="application/json")
@task(returns=str)
def post_with_param(payload):
    pass
```

Tasks' Prolog and Epilog

Define actions to be performed before and/or after the tasks

```
from pycompss.api.prolog import prolog
from pycompss.api.epilog import epilog

@prolog(binary="echo", args="just a prolog")
@epilog(binary="echo", args="just an epilog")
@task(return=1)
def basic():
    return True
```

Software tasks

Software enables the invocation of complex software with compact tasks definition

```
from pycompss.api.software import software

@software(config_file="/path/to/config/binary_basic.json")
@task()
def my_date_binary(d_prefix, param):
    pass
```

```
# binary_basic.json
{
  "execution": {
    "type": "binary",
    "runner": "mpirun",
    "binary": "date",
    "working_dir": "/tmp"
  }
}
```

Grouping

Tasks can be arranged in groups to allow partial barriers or cancel specific sets of tasks on failure

```
from pycompss.api.api import TaskGroup
from pycompss.api.api import compss_barrier_group

with TaskGroup('Group1', False):
    for i in range(NUM_TASKS):
        task1()
        task2()
    ...
...
compss_barrier_group('Group1')
```

Failure Management

On task failure: RETRY, CANCEL_SUCCESSORS, FAIL or IGNORE. To handle task failures, but also prune workflow or skip tasks

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

Task Timeout

Define a maximum execution time (in seconds) for a task

```
@task(file_path=FILE_IN, time_out=200)
def time_out_task(file_path):
    ...
```

I/O Intensive Tasks

I/O intensive tasks can share resources with regular tasks. A specific bandwidth to be used can be set

```
from pycompss.api.IO import IO

@constraint(storageBW = 9)
@IO()
@task()
def save(data, itr):
    ...
    f = open(dest, "w")
    f.write(data)
    f.flush()
    ...
```

```
@task(returns=list)
def compute():
    ...
    for i ...
  #some computation
    ...
```

```
def main():
    for j in range(N):
        for i in range(48):
            c[i] = compute()

    for i in range (48):
        save(c[i], i*(j+1))
```

Support for Data Streams

Integrate task-flow and data-flow tasks

```
@task(fds=STREAM_OUT)
def sensor(fds):
  ...
    while not end():
        data = get_data_from_sensor()
        f.write(data)
    fds.close()
```

```
@task(fds_sensor=STREAM_IN, filtered=OUT)
def filter(fds_sensor, filtered):
  ...
    while not fds_sensor.is_closed():
        get_and_filter(fds_sensor, filtered)
```

DDS

Distributed Data Set and operators' collection

```
from pycompss.dds import DDS

dataset_path = "/path/to/dataset/"
results = DDS()
        .load_files_from_dir(dataset_path)
        .flat_map(lambda x: x[1].split())
        .map(lambda x: "".join(e for e in x if e.isalnum()))
        .count_by_value(as_dict=True)
```

Containerised task

Encapsulate and execute a task. Can be combined with @binary, @ompss, @mpi, @Julia and @decaf

```
from pycompss.api.binary import binary

@container(engine="SINGULARITY", image="compss", options="-v
/home/user/mount_directory:/home/user/mount_directory"))
@task(returns=1, num=IN, in_str=IN, fin=FILE_IN)
def container_fun(num, in_str, fin):
    # Sample task body:
    with open(fin, "r") as fd:
        num_lines = len(fd.readlines())
    str_len = len(in_str)
    result = num * str_len * num_lines

    # You can import and use libraries available in the container

    return result
```

Multiple Implementations of a Task

Solve the same problem in different ways. Constraints help to select the proper method for a resource

```
from pycompss.api.implement import implement

@implement(source_class="sourcemodule", method="my_func")
@constraint(app_software="numpy")
@task(returns=list)
def myfunctionWithNumpy(list1, list2):
    # Operate with the lists using numpy
    return resultList

@task(returns=list)
def my_func(list1, list2):
    # Operate with the lists using built-int functions
    return resultList
```

Task Exceptions

Raise and treat exceptions that come from tasks. Can be used in combination with groups of tasks

```
from pycompss.api.exceptions import COMPSsException

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

```
from pycompss.api.api import TaskGroup

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

Checkpointing

Save the execution progress of the application

```
from pycompss.api.api import compss_snapshot

compss_snapshot()
```

In runcompss or enqueue_compss

```
--checkpoint_params=period.time:s,avoid.checkpoint:[checkpoint_file_test.increment]
--checkpoint=es.bsc.compss.checkpointer.policies.CheckpointPolicyPeriodicTime
--checkpoint_folder=/tmp/checkpointing/
```

Leveraging NUMBA

Just In Time compilation for Python code to accelerate the task execution

```
@task(returns=1, numba=True)
def ident_loops_jit(x):
    r = np.empty_like(x)
    n = len(x)
    for i in range(n):
        r[i] = np.cos(x[i]) ** 2 + np.sin(x[i]) ** 2
    return r
```

Integration with Persistent Memory

Objects can be declared persistent. The underlying framework handles object distribution and replication

```
a = SampleClass ()
a.make_persistent()
Print a.func (3, 4)

a.mytask()
compss_barrier()

o = a.another_object
```

Agents Deployment

Alternative deployment as autonomous Agents

```
$ compss_agent_start  --hostname=192.168.1.100 --classpath=/app/path.jar
$ compss_agent_call_operation --master_node="127.0.0.1" --
master_port="46101" --method_name="demoFunction"
es.bsc.compss.test.DemoClass 1
$ compss_agent_add_resources --agent_node=192.168.1.70 --agent_port=46101 -
-cpu=4 192.168.1.72 Port=46102
$ compss_agent_lost_resources [options] resource_name
```

Schedulers

Change the way tasks are selected for execution. Can be enhanced with input profiling. In runcompss or enqueue_compss:

```
--scheduler=<className>                 Class that implements the Scheduler
for COMPSs

--input_profile=<path>                  Path to the file which stores the
input application profile
```

Performance Analysis with Paraver

Get extreme level of detail of your COMPSs application runs, to analyse any issues or inefficiencies

```
$ enqueue_compss --tracing application_name application_args

$ compss_gentrace --trace_name=my_app_name
```

Application Prolog and Epilog

Define actions to be performed before and/or after the application. In runcompss or enqueue_compss:

```
--prolog="<string>"        Task to execute before launching COMPSs (Notice
the quotes). This argument can appear multiple times for more than one
prolog action

--epilog="<string>"        Task to execute after executing the COMPSs
application (Notice the quotes). This argument can appear multiple times
for more than one epilog action
```

Workflow Provenance (FAIR)

Publish and cite your execution results in scientific papers. Artifact for reproducibility and replicability. Debug your code

- **YAML** file describing the application and its authors (optional)
- **Submit with -p** or **-z** flag with runcompss or enqueue_compss
- **Resulting sub-folder or zip** with an artifact in **RO-Crate** format
- Debug your application with **pycompss inspect**
- The artifact can be uploaded to **Zenodo, WorkflowHub or ROHub** for **DOI** generation

Energy monitoring with EAR

Get power consumption during the execution to evaluate the application efficiency

```
$ enqueue_compss --ear=true application_name application_args

$ compss_gentrace_full
```

https://compss.bsc.es/

DT-GEO ICOS CyclOps caelestis CEEC ODISSEE Online Data Intensive Solutions for Science in the Exabytes Era HP2C-DT Colmena Generalitat de Catalunya Departament de Recerca i Universitats Ministerio de Ciencia e Innovación Financiado por la Unión Europea NextGenerationEU Plan de Recuperación, Transformación y Resiliencia Agencia Estatal de Investigación