ExaQUte

Exascale **Q**uantification of **U**ncertainties for **Te**chnology and Science Simulation

D4.1 Working Python API to schedule MLC routines

Document information table

Contract number:	800898
Project acronym:	ExaQUte
Project Coordinator:	CIMNE
Document Responsible Partner:	CIMNE
Deliverable Type:	Software
Dissemination Level:	Public
Related WP & Task:	WP 4, Task 4.1
Status:	Final version

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No **800898**

Authoring

Prepared by:				
Authors	Partner	Modified Page/Sections	Version	Comments
Ramon Amela	BSC		V0	Creation of the document
Rosa M. Badia	BSC	Sections 1, 2, 3 and 5	V0	Changes in some sections
Riccardo Rossi	CIMNE			
Stanislav Bohm	IT4i			
Jakub Beranek	IT4i			
Contributors				

Change Log

Versions	Modified Page/Sections	Comments

Approval

Aproved by:				
	Name	Partner	Date	OK
Task leader	Rosa M. Badia	BSC	27.7.18	OK
WP leader	Rosa M. Badia	BSC	27.7.18	OK
Coordinator	Riccardo Rossi	CIMNE	27.7.18	OK

Executive summary

This deliverable focuses on the definition of a common API for the PyCOMPSs programming model and HyperLoom scheduler provided respectively by BSC and IT4I. The objective of the work is to hide the details of the actual task scheduling technology, so that the Multi Level Monte Carlo Python engine is agnostic of the backend being employed.

It includes the description of:

- Common API for calls
- Examples of usage
- Basic Documentation

The document also contains an initial description on how MPI-distributed data shall be treated from the scheduling point of view.

Table of contents

List of Figures

List of Tables

Nomenclature / Acronym list

Acronym	Meaning
API	Application Programming Interface
ExaQUte	EXAscale Quantification of Uncertainties for Technology and Science Simulation
DAG	Directed Acyclic Graph
FILE_IN	Path to a file passed to a function that is not modified
FILE_INOUT	Path to a file passed to a function that is modified during the call
FILE_OUT	Path to a file passed to a function that is created during the call
HPC	High Performance Computing
IN	Parameter of a function that is not modified
INOUT	Parameter of a function that is modified during the call
OpenMP	Open Multi Processing
MPI	Message Passing Interface
PBS	Portable Batch System
PyCOMPSs	Python binding for COMPS Superscalar
SLURM	Simple Linux Utility for Resource Management

1 Introduction

PyCOMPSs and HyperLoom are task-based programming environment which enable the parallelization of sequential codes that can be then executed in distributed computing platforms.

State of the art description including references to relevant articles, e.g. [**? ? ? ? ?**]

2 API definition

The API has been designed in such a way that it is able to express the union of functionalities supported by both PyCOMPSs and HyperLoom. The aim of this section is to explain the API desirable properties and show the reasoning behind the definitions.

2.1 Common API

First of all, both frameworks has been analyzed to find the common points. All along the discussion, it has been taken into account that the users intended to use the API are not computer scientists. Hence, simplicity has been taken as a capital property to be achieved. In addition, the design tries to make as easy as possible the transformation of a sequential code into a distributed one.

2.1.1 Launch process

Both BSC and IT4I have already developed scripts that ease the deployment of the applications in HPC environments with queue systems. Hence, the user should be able to launch the application without managing all the information relative to the infrastructure. That is, being able to execute the code into different machines with a different amount of resources and a different architecture without changing it.

2.1.2 Distribution strategy

There are several distribution strategies. Nevertheless, both programming models define a workflow based on the *taskification* of the work. This means that some regions of code are defined to be executed remotely. The scheduler automatically detects the dependencies between tasks building the DAG that define the order in which the different tasks must be executed. In addition, generated objects are only brought to the master under demand.

2.1.3 API format

Considered the previous point, it has been decided to define the distributed portions of code with decorators. This strategy has several advantages. First of all, allows the user to define the main workflow sequentially, using a local definition of the decorator that avoids distributing the work. In addition, hides all the initialization process and avoids the direct communication between the user and the scheduler. Finally, makes possible to change from a sequential version to a distributed one o between both schedulers just changing the *module import*. This fact guarantees that the user can change the execution mode without modifying a single line of the code.

2.1.4 API calls

The most important functionalities shared between both programming models are the following ones:

- Task definition
- Get value to the master/client node
- Wait until all the tasks have finished their execution
- Delete object from the remote nodes

2.1.5 Task constraints

Both PyCOMPSs and HyperLoom allow the user to specify resource constraints. PyCOMPSs has a wider variety of resource characteristics that can be indicated by the user. On the other side, HyperLoom allows the user to define them inline a dynamic way.

2.2 PyCOMPSs functionalities

There are three main functionalities that has been added to the project since they were already supported by PyCOMPSs.

2.2.1 Files as a special type

A string passed to a function can be declared as a file. This fact implies that the Runtime copies the corresponding file to/between/from the worker nodes to ensure that a given task will find the file in the renamed path that the Runtime pass to it as parameter. The same behavior than in the *object* case is offered, that is FILE_IN and FILE_INOUT. In addition, it is possible to indicate FILE_OUT in case the file is generated into the task. This behavior does not make sense in the *object* case since either a reference to an already created object is passed to the function or a new instance is created and given back to the main program as return parameter. Finally, it has to be considered that this separate consideration implies the creation of the special call `delete_file` to delete all the available copies in the worker nodes.

2.2.2 Constraint decorator

It is possible to define constraints for each task. To this end, the decorator @*constraint* followed by the desired constraints needs to be placed over the @task decorator as shown in Figure 1.

This decorator enables the user to set the particular constraints for each task, such as the amount of Cores required explicitly. Alternatively, it is also possible to indicate that the value of a constraint is specified in a environment variable. Figure 2 shows how to express this constraints.

A full description of the supported constraints can be found in Table 1.

```
from pycompss.api.task import task
from pycompss.api.constraint import constraint
from pycompss.api.parameter import INOUT

%*{\bf @constraint }*)(ComputingUnits="4")
%*{\bf @task }*)(c = INOUT)
def func(a, b, c):
    c += a*b
    ...
```

Figure 1: Constraint basic usage

```
from pycompss.api.task import task
from pycompss.api.constraint import constraint
from pycompss.api.parameter import INOUT

%*{\bf @constraint }*)(ComputingUnits="4", AppSoftware="numpy,scipy,gnuplot",
    memorySize="$MIN_MEM_REQ")
%*{\bf @task }*)(c = INOUT)
def func(a, b, c):
    c += a*b
    ...
```

Figure 2: Constraint advanced usage

Field	Value type	Default value	Description
ComputingUnits	<string>	¨1¨	Required number of computing units
ProcessorName	<string>	¨[unassigned]¨	Required processor name
ProcessorSpeed	<string>	¨[unassigned]¨	Required processor speed
ProcessorArchitecture	<string>	¨[unassigned]¨	Required processor architecture
ProcessorType	<string>	¨[unassigned]¨	Required processor type
ProcessorPropertyName	<string>	¨[unassigned]¨	Required processor property
ProcessorPropertyValue	<string>	¨[unassigned]¨	Required processor property value
ProcessorInternalMemorySize	<string>	¨[unassigned]¨	Required internal device memory
-	List<@Processor>	¨{}¨	Required processors (check Table 2 for Processor details)
MemorySize	<string>	¨[unassigned]¨	Required memory size in GBs
MemoryType	<string>	¨[unassigned]¨	Required memory type (SRAM, DRAM, etc.)
StorageSize	<string>	¨[unassigned]¨	Required storage size in GBs
StorageType	<string>	¨[unassigned]¨	Required storage type (HDD, SSD, etc.)
OperatingSystemType	<string>	¨[unassigned]¨	Required operating system type (Windows, MacOS, Linux, etc.)
OperatingSystemDistribution	<string>	¨[unassigned]¨	Required operating system distribution (XP, Sierra, openSUSE, etc.)
OperatingSystemVersion	<string>	¨[unassigned]¨	Required operating system version
WallClockLimit	<string>	¨[unassigned]¨	Maximum wall clock time
HostQueues	<string>	¨[unassigned]¨	Required queues
AppSoftware	<string>	¨[unassigned]¨	Required applications that must be available within the remote node for the task

Table 1: Arguments of the *@constraint* decorator

All constraints are defined with a simple value except the *HostQueue* and *AppSoftware* constraints, which allow multiple values.

The *processors* constraint allows the users to define multiple processors for a task execution. This constraint is specified as a list of @Processor annotations that must be defined as shown in table 2

Annotation	Value type	Default value	Description
computingUnits	<string>	¨1¨	Required number of computing units
name	<string>	¨[unassigned]¨	Required processor name
speed	<string>	¨[unassigned]¨	Required processor speed
architecture	<string>	¨[unassigned]¨	Required processor architecture
type	<string>	¨[unassigned]¨	Required processor type
propertyName	<string>	¨[unassigned]¨	Required processor property
propertyValue	<string>	¨[unassigned]¨	Required processor property value
internalMemorySize	<string>	¨[unassigned]¨	Required internal device memory

Table 2: Arguments of the *@Processor* decorator

2.2.3 Implement decorator

PyCOMPSs allows the user to define several versions of the same tasks. The main idea behind this functionality is having several ways to perform the same operation. In the general case, the constraints of each version are different.
For example, it would be possible to have an implementation for an accelerator (GPU or FPGA) and an implementation using common CPUs. It would also be possible to define an implementation that uses `OpenMP` occupying a single node and a `MPI` version occupying several computing nodes.

Figure 3 a use example of this decorator. In this case, the secondary implementation uses a library that may be installed in a subset of the computing nodes. The information that must be provided is the class in which the main implementation is coded and the method name.

2.3 HyperLoom functionalities

2.3.1 Dynamic resource requests

HyperLoom allows to define resource request for each task individually.

```python
from pycompss.api.implement import implement

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

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

Figure 3: Implement usage example

2.3.2 Run external program with zero-copy execution

HyperLoom provides "run" method to running external programs. Data objects are managed via symlinking and moving on RAM disk. It avoids unnecessary copy of data and reading data by worker at all if computation stays on the same worker.

3 Methods

Both frameworks have to implement all the decorations and API calls described in this section even if they don't have any impact in the scheduling process. This fact guarantees that the code do not crash when switching between them. It has to be taken into account that Figures 4 and 5 contain the wrapper coded to work in local. This code helps understanding the expected behavior but does not have any scheduling implementation.

3.1 Decorators

3.1.1 Task decorator

First of all, a decorator has been defined to indicate the tasks. It's basic syntax is shown on Figure 4. All the functions marked with this decorator will be executed remotely.
By default it is assumed that all the function parameters are IN and there is no return value. In addition, there are two hints that must be given otherwise:

1. `returns` allows the user to say how many return values has the function.
2. `variable_name={INOUT, FILE_IN, FILE_OUT, FILE_INOUT}` makes possible to indicate either that the input object is modified or that the input string corresponds to a file. This way, the scheduler can transfer the file between the diferent worker nodes to make the execution possible.

Finally, it has to be taken into account that there is an optional parameter associated to the keyword `scheduling_constraints` that makes possible to pass an instance of `ExaquteExecutionConstraints` that allows the user to specify task constraints in a dynamic way. That is, in execution time. This point has to be discussed more deeply considering the users feedback. Even if it allows them more expresivity, introduces scheduling dependent code into de user's code, fact that was agreed to avoid at the maximum. Alternatively, it would be possible to define several versions of the same call with different execution constraints.

3.1.2 Constraint decorator

Considering that this is a functionality that BSC had already implemented, the syntax remains exactly as presented on subsection 2.2. More keywords could be added considering the user's demands and the project's particularities.

3.1.3 Implements

This case is analogous to the previous point. Hence, the same notation already implemented on PyCOMPSs has been kept.

3.2 API calls

Figure 5 shows the basic definition of the api calls implemented. The following points summary their basic expected behavior:

- `barrier()`
  The purpose of this call is to wait until all the tasks have been executed.
- `get_value_from_remote(obj)`
  This API call brings back to the master/client the value of a remote computed object.
- `delete_object(obj)`
  This API call removes all the copies allocated in the workers of the given object.
- `delete_file(path_to_file)`
  This API call removes all the copies allocated in the workers of the given file.
- `compute(value)`
  This API call submits the given value to the scheduler, starting the computation considering all the DAG generated until the moment.

4 Example of usage

This example, even is is a toy code that does not perform any useful computation, has the property of testing all the functionalities described in section 3. Figure 6 shows how the example code launches the initialization of several models with really variate granularity in distributed nodes. This generated model is defined in Figure 8. The most important think to realize that it contains the `Kratos` model that will have to be handled in the real project. In addition, an intermediate class has been defined as shown in Figure 7 to show how distributed computations can be hidden in an intermediate class to ease the workflow definition. All the distributed functions have been defined in Figure 7 to ease the identification of the parts that will be executed in the worker nodes. The interactions with the scheduler are completelly contained in this file. Hence, changing from local to distributed and between both frameworks is done changing a single import.

5 Data requirements

Concerning the data requirements, it is possible to define two different project phases. Initially, data will be serialized to a single string. Next, distributed objects will be handled.

5.1 Single serialized object

In the first stage, medium size tasks will be launched in such a way that all the data can be fit in the memory of a single node. All the data passed to a task as a parameter must be serializable. This fact implies the definition of the following functions:

1. save
   It is possible to define this functions in two different ways.
   First of all, the function could have the following parameters:

   - obj: input object to serialize
   - return: a string or a stream in which to write the serialized object

   Otherwise, an alternative definition is also possible:

   - obj: input object to serialize
   - path: input string representing the path into which a file containing the serialized object should be created

2. load
   Afterwards, the serialized objects must be recoverable. This implies providing the symmetrical functionalities to the ones defined previously.
   In case the first option has been chosen, the following parameters are involved into the function call:

   - str: input string or stream containing the serialized object
   - return: original object passed to the *save* function

   The second version of this function should have the following parameters:

   - path: input string representing the path of the file containing the serialized object
   - return: original object passed to the *save* function

5.2 Distributed serialized object

In a more advanced stage, *mpi* simulations will be run, occupying several computing nodes. The case in which a single node memory is not capable to store all the data handled by each task is contemplated. Hence, a single serialized string/file is no longer possible. There is a deliverable in the 18th month concerning this problematic. Nevertheless, the discussion has already started. Some desirable properties have already been defined:

- The objects should be serialized in a distributed way, meaning that each *MPI* process should be responsible to the serialization and later recovery of its piece of data

- Since there is no longer a single serialization for each object, two main strategies are possible to express the dependencies between tasks:

    1. Define a component into the programming model that can be directly called from the mpi processes in such a way that it tracks all the serialized model parts and its location, being in charge to store it in a file in the early stages and in memory in more advanced phases
    2. Not considering the whole model at all and define all the dependencies in function of the several parts. In this case, the scheduler should be aware of the *MPI* process associated to each dependency in order to perform the transferences and set all the *MPI* environment variables accordingly.

Both solutions succeed in avoiding the shared file system and the serialization of the whole model into a single serialization. However, the first option seems more interesting as it delegates all the data handling part to the programming models. It keeps better the philosophy of having as less code as possible related to its distribution.
Nevertheless, it is important to keep in mind that this phase is still in a really early stage. Some discussion between all the involved partners needs to be carried out. A consensus should be achieved in such a way that the desired properties are achieved and the solution fits as well as possible in the already implemented codes.

```
class Exaqute_task(object):

   def __init__(self, *args, **kwargs):
      pass

   def __call__(self, f):
      def g(*args, **kwargs):
         if "scheduling_constraints" in kwargs:
            del kwargs["scheduling_constraints"]
            return f(*args, **kwargs)
      return g

```

Figure 4: Decorator definition for the local case

```
def barrier():
   pass

def get_value_from_remote(obj):
   return obj

def delete_object(obj):
   del obj

def delete_file(file_path):
   import os
   os.remove(file_path)

def compute(obj):
   return obj

```

Figure 5: API calls definition for the local case

References

A Methods

B Example of usage

```
import sys
import MultilevelMonteCarloFunctions
from MultilevelMonteCarloStep import MultilevelMonteCarloStep

short = False

if short:
   multiplier_low = 1
   multiplier_up = 2
else:
   multiplier_low = 13
   multiplier_up = 20

if __name__ == "__main__":

   models_path = sys.argv[1]

   steps = MultilevelMonteCarloFunctions
      .generate_models(models_path, 3, [], short, multiplier_low, multiplier_up)

   while len(steps) > 0:
      runs = []
      comparaisons = []
      versions = []
      for i in xrange(len(steps)):
         runs.append([])
         if i > 0:
            comparaisons.append([])
         for j in xrange(steps[i][1].amount_executions):

            current_step_model = MultilevelMonteCarloStep(steps[i][0] + ".mdpa")

            steps[i-1][1].cpus_per_task = 2
            versions.append(current_step_model.generate_random(scheduling_
    constraints = steps[i-1]))

            current_step_model.run()

            runs[i].append(current_step_model)
            if i > 0:
               comparaisons[i-1].append(MultilevelMonteCarloFunctions.compute(runs[i
    ][j].compare_models(runs[i-1][j])))

      if(len(steps) > 1):
         partial_list = []
         for compare_list in comparaisons:
            partial_list.append(MultilevelMonteCarloFunctions.reduce_list(compare_
    list))

         verification_value = MultilevelMonteCarloFunctions.reduce_list(partial_list
    )

         verification_value = MultilevelMonteCarloFunctions.get_value(verification_
    value)

         versions_count = MultilevelMonteCarloFunctions.count_versions(*versions)

         versions_count = MultilevelMonteCarloFunctions.get_value(versions_count)

         print("VERIFICATION VALUE: " + str(verification_value.params))
         print("AMOUNT OF DIFFERENT VERSIONS USED: " + str(versions_count))

      else:
         verification_value = 0

      multiplier_low /= 2
      multiplier_up /= 2
      if short:
         steps = MultilevelMonteCarloFunctions
            .generate_models(models_path, 2, verification_value, short, multiplier_
    low, multiplier_up)
      else:
         steps = MultilevelMonteCarloFunctions
            .generate_models(models_path, 3, verification_value, short, multiplier_
    low, multiplier_up)

```

Figure 6: Basic use example

```
import MultilevelMonteCarloFunctions

class MultilevelMonteCarloStep(object):

    def __init__(self, model):
        self.model = MultilevelMonteCarloFunctions.load_model(model)
        self.params = None

    def generate_random(self, scheduling_constraints = None):
        self.params = MultilevelMonteCarloFunctions.generate_random(self.model,
            scheduling_constraints = scheduling_constraints)
        return self.params

    def run(self):
        MultilevelMonteCarloFunctions.run(self.model, self.params)

    def compare_models(self, modelB):
        return MultilevelMonteCarloFunctions.compare_models(self.model, modelB.model)

```

Figure 7: Monte Carlo step definition

```
import KratosMultiphysics
import KratosMultiphysics.FluidDynamicsApplication

class MultilevelMonteCarloModel(object):

    def __init__(self, path_to_model = None):
        if(not path_to_model is None):
            model_part_name = "MainRestart"
            model_part = KratosMultiphysics.ModelPart(model_part_name)
            model_part.AddNodalSolutionStepVariable(KratosMultiphysics.DISPLACEMENT)
            model_part.AddNodalSolutionStepVariable(KratosMultiphysics.VISCOSITY)
            model_part_io = KratosMultiphysics.ModelPartIO(path_to_model)
            model_part_io.ReadModelPart(model_part)
            self.model_part = model_part

def set_model_part(self, model_part):
    self.model_part = model_part

```

Figure 8: Model definition

```
import random
from MultilevelMonteCarloParams import MultilevelMonteCarloParams
from MultilevelMonteCarloModel import MultilevelMonteCarloModel
import KratosMultiphysics
import math
from subprocess import Popen, PIPE
from ExaquteExecutionConstraints import ExaquteExecutionConstraints

from ExaquteTaskPyCOMPSs import *

def _listdir_shell(path, *lsargs):
    p = Popen(('ls', path) + lsargs, shell=False, stdout=PIPE, close_fds=True)
    return [path.rstrip('\n') for path in p.stdout.readlines()]

@ExaquteTask(path_to_model = FILE_IN, returns = 1)
def load_model(path_to_model):
    return MultilevelMonteCarloModel(path_to_model[:-5])

@constraint(computingUnits = "2")
@ExaquteTask(model = INOUT, returns = 1)
def generate_random(model):
    model_part = model.model_part
    # generate random number
    for node in model_part.Nodes:
        rand = random.random()
        node.SetSolutionStepValue(KratosMultiphysics.VISCOSITY, 0, rand)
    params = MultilevelMonteCarloParams(1)
    return params

@implement(source_class="MultilevelMonteCarloFunctions", method="run")
@ExaquteTask(model = INOUT, params = INOUT)
def run_2(model, params):
    model_part = model.model_part
    for node in model_part.Nodes:
        node.SetSolutionStepValue(KratosMultiphysics.VISCOSITY, 0,
              node.GetSolutionStepValue(KratosMultiphysics.VISCOSITY, 0) + 1.0)
    params.set_value(2)

@ExaquteTask(model = INOUT, params = INOUT)
def run(model, params):
    model_part = model.model_part
    for node in model_part.Nodes:
        node.SetSolutionStepValue(KratosMultiphysics.VISCOSITY, 0,
              node.GetSolutionStepValue(KratosMultiphysics.VISCOSITY, 0) + 1.0)
        params.set_value(1)

@ExaquteTask(returns = 1)
def compare_models(model_1, model_2):
    model_part_1 = model_1.model_part
    model_part_2 = model_2.model_part
    # take the average
    avgModel = 0.0
    n = max(1, len(model_part_1.Nodes) + len(model_part_2.Nodes))
    for node in model_part_1.Nodes:
        avgModel += node.GetSolutionStepValue(KratosMultiphysics.VISCOSITY)
    for node in model_part_2.Nodes:
        avgModel += node.GetSolutionStepValue(KratosMultiphysics.VISCOSITY)
    avgModel /= n
    return MultilevelMonteCarloParams(avgModel)

def get_value(obj):
    return get_value_from_remote(obj)

@ExaquteTask(returns = 1)
def reduce_function(*reduce_list):
    sum_list = reduce_list[0].params
    for elem in reduce_list[1:]:
        sum_list += elem.params
    avg = sum_list / len(reduce_list)
    return MultilevelMonteCarloParams(avg)

@ExaquteTask(returns = 1)
def count_versions(*versions):
    from collections import Counter
    int_list = []
    for elem in versions:
        int_list.append(elem.params)
    ret = dict(Counter(int_list))
    return ret

```

Figure 9: Function definition