ExaQUte

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

D5.1 ExaQUte API for MLMC

Document information table

Contract number:	800898
Project acronym:	ExaQUte
Project Coordinator:	CIMNE
Document Responsible Partner:	TUM
Deliverable Type:	Report, Other
Dissemination Level:	Public
Related WP & Task:	WP5, Task 5.1
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
Daniel Drzisga	TUM			
Mario Teixeira Parente	TUM			
Roland Wüchner	TUM			
Contributors				

Change Log

Versions	Modified Page/Sections	Comments

Approval

Approved by:				
	Name	Partner	Date	OK
Task leader	Roland Wüchner	TUM	30.7.18	OK
WP leader	Fabio Nobile	EPFL	30.7.18	OK
Coordinator	Riccardo Rossi	CIMNE	30.7.18	OK

Executive summary

This deliverable focuses on the design of an interface between MLMC algorithms, the scheduling engine, and problem solvers. For this purpose an API definition is proposed together with a basic reference implementation in Python. This work serves as a first step for the development of the ExaQUte MLMC Python engine.

It includes:

- API definition
- Demonstrator code and description
- Example of usage

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
QoI	Quantity of Interest
MC	Monte Carlo
MLMC	Multilevel Monte Carlo method
C-MLMC	Continuation Multilevel Monte Carlo method
HPC	High performance computing
PDE	Partial differential equation

1 Introduction

In this report, we give a brief introduction to the MLMC method and its main goals; we also provide a first API definition of the MLMC Python engine in Section 2 and a reference implementation for a test problem in Section 3.

The *Multilevel Monte Carlo method* (MLMC) [1, 5, 6, 8] is a technique to reduce computational cost for quantifying uncertainties of a random output *Quantity of Interest* (QoI) of a complex computational model. It estimates statistics of the QoI like e. g., its expected value. Let Q denote the output QoI which is random due to randomness in the input parameter of the model. We are interested in computing, or more precisely, approximating $\mathbb{E}[Q]$. MLMC is flexible since it does not require regularity of the parameter-to-QoI map and furthermore breaks the curse of dimensionality, i. e., its performance does not depend on the number of input random parameters. MLMC accelerates convergence of estimators of $\mathbb{E}[Q]$, with respect to standard Monte Carlo (MC) estimators, which are often unaffordable for complex computational models. Also, it is well-suited for parallelization in high performance computing (HPC) contexts [3].

Informally, MLMC distributes computational work on different levels from a hierarchy of different accuracies, e. g,. a hierarchy of meshes for approximating solutions of partial differential equations (PDEs). A crucial decision using MLMC is *how much* work is allocated to each level and *how* the overall error is split into different contributions. Standard MLMC splits the overall error into two equal parts, related to the discretization of the PDE and the statistical errors of the level-wise MC estimators, although different splittings are also possible [7].

Also, the strategy of distributing the work over the levels relies on approximations of variances, which can be expensive to compute.

There exist adaptive versions of MLMC that choose optimal values for the number of levels and the computational effort on each level [2–4, 9]. It is obvious that the cost for computing optimal values should not exceed the overall cost of the resulting MLMC estimator. A recent variant of MLMC, tackling these issues, is the *Continuation Multilevel Monte Carlo method* (C-MLMC) [2, 9], which uses an algorithm to learn parameters for distributing the work and splitting the overall error accordingly, in order to save as much cost as possible while guaranteeing a final error within a prescribed tolerance.

2 API definition

In this section, we propose an API for the development of an ExaQUte MLMC Python engine. For this purpose, we follow a modular approach where the MLMC algorithm is completely decoupled from the problem to be solved. This allows for rapid application development of different MLMC strategies without having to change anything in the problem solver and vice versa. The presented API and the reference implementation in Section 3 are only a first proposal and subject to change in the future.

2.1 Problem Abstraction

In order to decouple the problem of interest from the employed MLMC algorithm, we propose the following two interfaces. First, the *MLMC Algorithm interface* is described in Section 2.1.1 and then the *Problem interface* is described in Section 2.1.2.

Figure 1 shows a schematic view of our API where the MLMC algorithm spawns new solver processes for each required sample and after a task is finished, receives the corresponding QoI of a sample. In this particular example, solvers on three levels are running in parallel. Solvers on finer levels require more computational resources, i.e., more memory and larger number of processors. Since the goal of the MLMC method is to do most of the work on coarse levels, most of the samples are only required to be computed on the coarse levels.

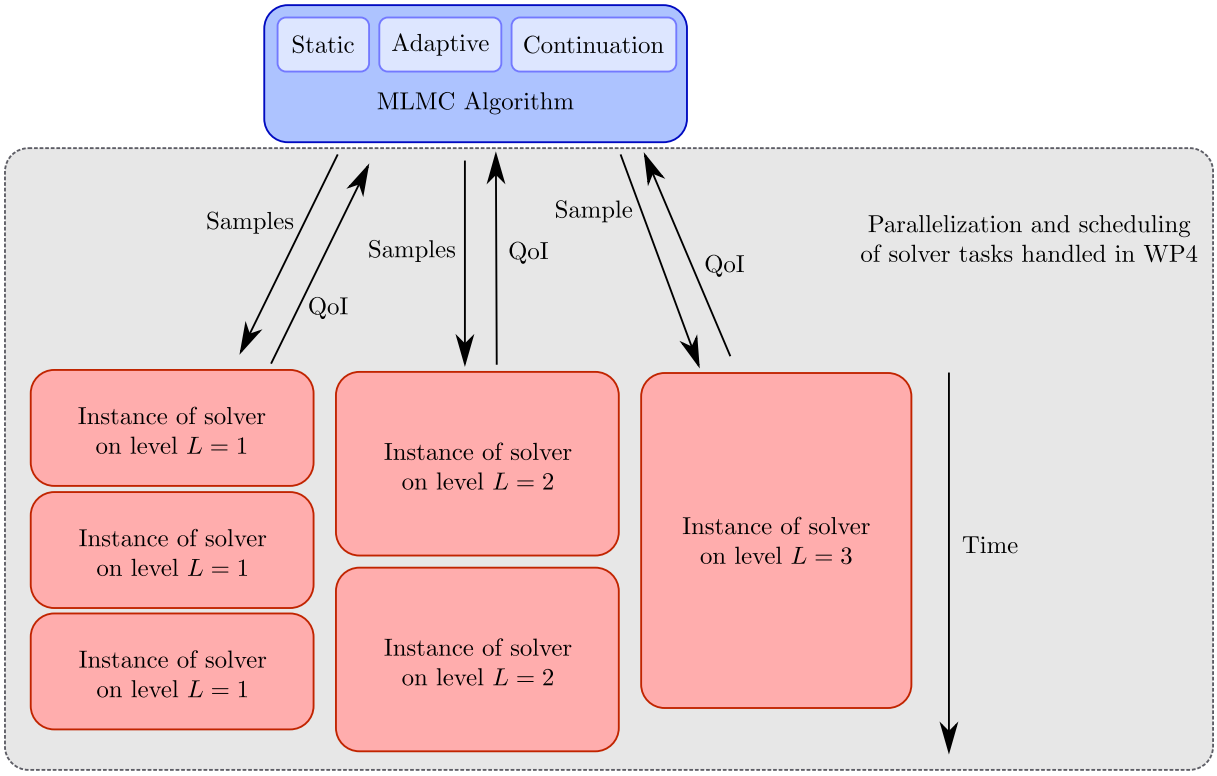

Figure 1: Schema of our MLMC API with three solver levels running in parallel.

2.1.1 MLMC Algorithm interface

Implementations of the MLMC algorithm interface are the core of each MLMC simulation (cf. blue box in Figure 1). A particular implementation is responsible for the distribution of samples for each level and the computation of statistics of the QoI. As described in Section 1, it is possible to implement different strategies and we mainly focus on static, adaptive, and continuation strategies.

Each instance of the MLMC algorithm requires information on the problem which needs to be solved. This is done through the *Problem interface* described in Section 2.1.2. This interface provides a method that returns the QoI obtained on a particular level. At each MLMC iteration, the algorithm decides how many samples are required on each level such that a given accuracy is obtained. The samples are then passed to the scheduler which is then responsible for the distribution of tasks depending on the required resources (cf. gray box in 1). After a task finishes, the MLMC algorithm receives its QoI. Depending on the MLMC strategy and the problem, further iterations may be necessary before the algorithm converges.

2.1.2 Problem interface

The *Problem interface* encapsulates information of the problem of interest, its QoI, and the underlying solver. It provides a method which returns the QoI obtained on a specific level. This method is called either by the MLMC algorithm directly or its attached scheduler for each sample. In our typical cases, calling this method involves solving a PDE with a randomly sampled coefficient on a refined mesh.

The red boxes in Figure 1 show different instances of *Problem interface* implementations running in parallel. It is important that the implementations of this interface must be thread-safe and may not interfere with other instances when executed in parallel.

Moreover, the solvers themselves may run in parallel which has to be taken into account by the task scheduler, so that the provided computational resources are efficiently utilized.

3 Demonstrator code

For demonstration purposes, we attached a Python source code that implements the suggested interfaces from Section 2 for a simple elliptic test problem. The MLMC algorithm interface is currently only implemented by an implementation of the C-MLMC algorithm. Furthermore, only a serial scheduling of solver tasks is considered. The scheduler will be replaced by a dynamic scheduler in the future, as it is described in WP4.

3.1 MLMC Algorithm implementation

The MLMC core of the demonstrator code is defined in `mlmc_routines/cmlmc.py`. It implements the *MLMC algorithm interface* presented in Section 2.1.1 by employing the C-MLMC method (cf. Figure 4). This C-MLMC implementation additionally depends on two files: The file `mlmc_routines/mlmc_level.py` contains the MC sampler for each level and the file `mlmc_routines/sample_moments.py` is required for the computation of statistics of the QoI and output, as well as for the update of C-MLMC parameters.

Instantiations of the core class require two parameters: A settings class encapsulating all required parameters of the MLMC strategy (cf. Figure 2) and an implementation of the *Problem interface*; see Section 3.2 for an example.

The extension of existing strategies and the implementation of new strategies is possible without any changes required in the particular problems and solvers.

3.2 Elliptic benchmark-problem implementation

The class `ellipt_2d` (cf. Figure 3) implements the *Problem interface* described in Section 2.1.2. It is found in the file `benchmark_problems.py` and implements the discretization, solver, and QoI computation of the following elliptic benchmark problem in 2D [8, Section 5.2]:
Let $D := [0,1]^2$. Find $u \in C^2(D) \cap C(\partial D)$ such that

$$\begin{aligned} -\Delta u(x,y) &= \xi\, f(x,y) && \text{in } D, \\ u(x,y) &= 0 && \text{on } \partial D, \end{aligned}$$

where $f(x,y) := -432x(x-1)y(y-1)$ and $\xi \sim \text{Beta}(2,6)$. The QoI is defined by

$$Q := \int_D u(x,y)\,dxdy.$$

It is important to note that this problem is a true benchmark problem, for which the exact expected value of Q is given by

$$\mathbb{E}[Q] = \frac{1}{4}\int_D u_1(x,y)\,dxdy,$$

where u_1 is the solution of the problem for $\xi = 1$. Thus only the solution of a single problem on a fine mesh is required to obtain a reference value of $\mathbb{E}[Q]$. This is very useful to verify the implementation and assess the algorithm's performances.

The problem is discretized by finite differences on a sequence of uniform grids and the discretized systems are solved using the *NumPy* Python package for scientific computing. The source code of the solver itself can be found in
`mlmc_routines/ellipt_2d/solver_ellipt.py`.

Changing the discretization of the problem or the solver is easily possible and does not require any changes in the used MLMC strategy.

4 Example of usage

Before running the demonstrator code, the following packages need to be installed on the system: Python 2 and the appropriate *NumPy* version. The demonstrator code may then be run by executing the attached file `main_ellipt.py` (cf. Figure 5).

A possible output in the terminal is presented in Table 1. It shows the evolution of the number of samples on each level during nine adaptation steps. As desired, the algorithm put more samples, and hence computational work, on coarser grids, whereas only a small amount of samples was distributed to finer grids.

	$L=1$	$L=2$	$L=3$	$L=4$	$L=5$	$L=6$	$L=7$	$L=8$
1	25	25	25	0	-	-	-	-
2	25	25	25	6	0	-	-	-
3	36	25	25	6	6	0	-	-
4	47	25	25	6	6	6	0	-
5	83	25	25	6	6	6	6	-
6	307	25	25	6	6	6	6	-
7	313	25	25	6	6	6	6	-
8	403	31	25	6	6	6	6	-
9	493	37	25	6	6	6	6	0

Table 1: Number of samples per level L adaptively chosen in nine steps.

Depending on the used MLMC strategy and solver, additional output details may be available. These may be helpful in the analysis of the efficiency of the MLMC strategy. In this particular demonstration, the following files are written to the file system: The file

`simulation_io/reports/P1` contains estimated errors and fitted rates for the C-MLMC algorithm, whereas the file `simulation_io/reports/P1_val` contains the obtained values of the required statistics of the QoI at each C-MLMC iteration.

References

[1] K. A. Cliffe, M. B. Giles, R. Scheichl, and A. L. Teckentrup. Multilevel monte carlo methods and applications to elliptic pdes with random coefficients. *Computing and Visualization in Science*, 14(1):3, 2011.

[2] N. Collier, A.-L. Haji-Ali, F. Nobile, E. von Schwerin, and R. Tempone. A continuation multilevel Monte Carlo algorithm. *BIT Numerical Mathematics*, 55(2):399–432, 2015.

[3] D. Drzisga, B. Gmeiner, U. Rüde, R. Scheichl, and B. Wohlmuth. Scheduling massively parallel multigrid for multilevel Monte Carlo methods. *SIAM Journal on Scientific Computing*, 39(5):S873–S897, 2017.

[4] D. Elfverson, F. Hellman, and A. Målqvist. A multilevel Monte Carlo method for computing failure probabilities. *SIAM/ASA Journal on Uncertainty Quantification*, 4(1):312–330, 2016.

[5] M. B. Giles. Multilevel Monte Carlo path simulation. *Operations Research*, 56(3): 607–617, 2008.

[6] M. B. Giles. Multilevel Monte Carlo methods. *Acta Numerica*, 24:259–328, 2015.

[7] A.-L. Haji-Ali, F. Nobile, E. von Schwerin, and R. Tempone. Optimization of mesh hierarchies in multilevel Monte Carlo samplers. *Stoch. PDEs: Anal. & Comp.*, 4(1): 76–112, 2016.

[8] M. Pisaroni, S. Krumscheid, and F. Nobile. Quantifying uncertain system outputs via the multilevel Monte Carlo method – Part I: Central moment estimation. MATHICSE Technical Report 23.2017, École Polytechnique Fédérale de Lausanne, 2017.

[9] M. Pisaroni, F. Nobile, and P. Leyland. A Continuation Multi Level Monte Carlo (C-MLMC) method for uncertainty quantification in compressible inviscid aerodynamics. *Computer Methods in Applied Mechanics and Engineering*, 326:20–50, 2017.

A Snippets

```
class simulation_ml(object):

    # sets tolerances and parameters for the cmlmc algorithm

    def __init__(self):

        # Tolerance and Confidences
        self.conf = False

        if self.conf is True:
            confidence = 0.90                    # Confidence Interval
            self.calpha = norm.ppf(confidence)
        else:
            self.calpha = 1.0

        self.type_ml = 'cmlmc'

        if self.type_ml == 'cmlmc':
            #self.theta  = 0.5        # Minimum Splitting parameter
            self.k0     = 0.1        # Certainty Parameter 0 rates
            self.k1     = 0.1        # Certainty Parameter 1 rates
            self.r1     = 1.25       # Cost increase first iterations C-MLMC
            self.r2     = 1.15       # Cost increase final iterations C-MLMC
            self.tol0   = 0.25       # Tolerance iter 0
            self.tolF   = 0.1        # Tolerance final
            self.N0     = 25
            self.L0     = 2

        else:
            print "only cmlmc available in this version"

        # Parallelization Settings
        self.uq_evaluation = 'serial'

```

Figure 2: Example MLMC settings class encapsulating all the required MLMC parameters for a particular MLMC strategy. In this case the C-MLMC algorithm.

```
class ellipt_2d(object):
    def __init__(self):
        self.name = "ellipt_2d"
        self.prob_path = path + '/mlmc_routines/' + self.name
        self.solverDET = 'solver_ellipt'
        self.solverMlevel  = 'mlmc_level'
        self.input_folder = None

        # Input Uncertainties
        self.input_rv_def = np.array([['Z',  'B', [], None ]])
        self.UNC_DEF = random_inputs(self.input_rv_def)

        # Output QoIs
        self.x_ref = np.arange(0,1,0.001)
        self.x_field = [self.x_ref, self.x_ref]

        # [xref_compute, xref_plot, k_reg, smooth, m_der, tau_cvar]
        self.CDF_GRD1 = [np.linspace(-1, 7, 1000), np.linspace(0, 6, 500),     10,
        0.0, 1,     0.95]

        self.QoI_def1    = np.array([['P1', 's', [2], [2], 'absolute' , self.CDF_
    GRD1],
                                     ['P2', 's', [2], [], None, None]])

        self.QoI_def     = np.array(self.QoI_def1[0,:],  ndmin=2)

        # MLMC Hierarchy
        self.Lmax     = 50

    def Nf_law(self,lev):
        # refinement strategy:
        # uniform mesh on level lev with h_lev=(1/N0)*2^(-lev)
        N0 = 5.
        M  = 2.
        NFF = (N0*np.power(M,lev))
        Nf2 = NFF**2

        return [NFF, Nf2]

```

Figure 3: Example implementation of the problem interface. It encapsulates information of the problem, the solver, as well as the QoI.

```
def mlmc(sim_ml, problem):
    mlmc_lev = importlib.import_module(problem.solverMlevel)
    print 'START'
    CMLMC_sim = CMLMC_simulation(sim_ml, problem)
    QOI_cmlmc = QOI_class(problem.QoI_def)

    # Nest
    if np.isscalar(sim_ml.N0):
        CMLMC_sim.Nest = sim_ml.N0*np.ones(sim_ml.L0+1)
    else:
        CMLMC_sim.Nest = sim_ml.N0

###############  Compute with an initial hierarchy ############################

    for level in range(sim_ml.L0+1):

        CMLMC_sim.level = level
        # RUN THE HIERARCHY
        mlmc_level = mlmc_lev.mlmc_l(level, CMLMC_sim.Nest[level], problem, sim_ml)

        if mlmc_level.bad_candidate is True:
            CMLMC_sim.conver = None
            CMLMC_sim.Nest   = None


            mlmc_rep.MEAN    = [0, 1, 0, 1]
            mlmc_rep.VAR     = [0, 1, 0, 1]
            print '#################### Det solver NOT-CONVERGED #############'
            return mlmc_rep

        # Update
        CMLMC_sim, QOI_cmlmc =  update_lev(CMLMC_sim, QOI_cmlmc, mlmc_level, problem
    .QoI_def)

########################## IMPOSE TOLERANCE ###################################

    # Compute Reference values, Tolerances and iter C-MLMC
    CMLMC_sim, QOI_cmlmc = set_tol(CMLMC_sim, QOI_cmlmc, problem.QoI_def)
    print CMLMC_sim.iE_cmlmc

######## STEP 2: Estimate problem parameters for Bayesian VAR #################
    CMLMC_sim, QOI_cmlmc = LS_rates(CMLMC_sim, QOI_cmlmc, problem.QoI_def)

    CMLMC_sim = var_estim(CMLMC_sim, range(0,CMLMC_sim.level+1))

    write_report(CMLMC_sim, QOI_cmlmc, problem.QoI_def, report_folder)

###############################################################################
    CMLMC_sim.iter = 1

    while CMLMC_sim.conv is not True:

        # Compute Tolerance for the iteration i
        CMLMC_sim = TOL_model(CMLMC_sim)

        # Compute Optimal Number of Levels
        CMLMC_sim = compute_levels(CMLMC_sim)

        if CMLMC_sim.ratesLS[1]<0.01 or CMLMC_sim.ratesLS[3]<0.01:
            CMLMC_sim.levelOPT = CMLMC_sim.levelOPT +1

        # Compute new Theta Spliting
        CMLMC_sim = THETA_model(CMLMC_sim, CMLMC_sim.levelOPT)

        if CMLMC_sim.theta>0 and CMLMC_sim.levelOPT <= CMLMC_sim.lmax:
        ######### STEP 4: Find Ml according to eq VAR and Theta ###############

            CMLMC_sim = optimal_Nsamp(CMLMC_sim)
            print CMLMC_sim.Nsam

        ######### Run new hierarchy using optimal levels and Nl ##########
            for l in range(0, CMLMC_sim.levelOPT+1):

                CMLMC_sim.level = l

                if CMLMC_sim.dNsam[CMLMC_sim.level] > 0:

                    mlmc_level = mlmc_lev.mlmc_l(CMLMC_sim.level, CMLMC_sim.dNsam[
    CMLMC_sim.level], problem, sim_ml)

                    CMLMC_sim, QOI_cmlmc = update_lev(CMLMC_sim, QOI_cmlmc, mlmc_
    level, problem.QoI_def)

            CMLMC_sim.level = CMLMC_sim.levelOPT

```

Figure 4: Example implementation of the MLMC core interface. In this particular case, the C-MLMC algorithm is implemented. (part 1)

```
            ######## Estimate problem parameters for Bayesian update ###############
            CMLMC_sim, QOI_cmlmc = LS_rates(CMLMC_sim, QOI_cmlmc, problem.QoI_def)

            CMLMC_sim, QOI_cmlmc = compute_errors(CMLMC_sim, QOI_cmlmc, problem.QoI_
    def)

            write_report(CMLMC_sim, QOI_cmlmc, problem.QoI_def, report_folder)

            if CMLMC_sim.iter >= CMLMC_sim.iE_cmlmc:
                # Check if the two error models are consistent
                TERR_diff = np.abs(CMLMC_sim.Terr_mod_STD-CMLMC_sim.Terr_sampl)
                SERR_diff = np.abs(CMLMC_sim.Errors[2]-CMLMC_sim.Errors[3])

                delta_t = 0.1

                if TERR_diff < delta_t * CMLMC_sim.tolF or SERR_diff < delta_t *
    CMLMC_sim.tolF:
                    print 'Err Mod Consistent'

                    if CMLMC_sim.Terr_sampl < CMLMC_sim.tolF: #and CMLMC_sim.Errors
    [2] < CMLMC_sim.theta*CMLMC_sim.tolF:
                        print '#################### CONVERGED ####################'
                        CMLMC_sim.conv = True

            CMLMC_sim.iter += 1

        else:
            CMLMC_sim.conv = True
            print '#################### MLMC NOT-CONVERGED ####################'
#            Mout = [0, 1, 0, 1]
#            Vout = [0, 1, 0, 1]

            return CMLMC_sim, QOI_cmlmc


    return CMLMC_sim, QOI_cmlmc

```

Figure 4: Example implementation of the MLMC core interface. In this particular case, the C-MLMC algorithm is implemented. (part 2)

```
sim_ml  = simulation_ml()
problem = ellipt_2d()

import cmlmc as mlmc
mlmc.mlmc(sim_ml, problem)

```

Figure 5: Example program where an MLMC settings and a Problem class are instantiated and the MLMC algorithm is executed. The MLMC algorithm internally instantiates problem solver processes with respective mesh resolutions depending on the underlying MLMC strategy. In this particular example, the C-MLMC algorithm is used.